FEDERAL UNIVERSITY OF RIO GRANDE DO SUL
INSTITUTE OF INFORMATICS

LEONARDO HAX DAMIANI

Fine Carrier Synchronization Unit for a Turbo Synchronization System

Computer Engineering Graduation Work.

Prof. Dr. Alexandre Carissimi
Advisor

Dipl.-Math. Uwe Wasenmüller
Co-advisor

Porto Alegre, june 2012.

Acknowledgments

I would like to express my special thanks of gratitude to my advisor Prof. Dr. Alexandre Casissimi. Throughout my thesis-writing period, he provided encouragement, sound advice, good teaching and a lot of virtuous ideas.

I am thankful to Prof. Dr.-Ing. Norbert Wehn and all the researchers from the Microelectronic Systems Design Research Group from University of Kaiserslautern, specially my co-advisor Dipl. Math. Uwe Wasenmüller, for his restless guidance, inspiration and determination in carrying out this project.

The respect to the secretaries, librarians and employees in the Informatics and Engineering departments of UFRGS and TUKL must be mentioned, for helping the departments to run smoothly and for assisting me in many different ways.

I wish to thank my entire family and friends for providing a loving and comprehensive environment for me. Last, and most important, I wish to thank my parents, Vilmar Damiani and Cleonice Hax Damiani. They bore me, raised me, supported me, taught me, and loved me.

SUMMARY

LIST OF ABBREVIATIONS AND ACRONYMS

GSM	Global System for Mobile
BER	Bit Error Rate
FER	Frame Error Rate
SNR	Signal to Noise Ratio
QPSK	Quadrature Phase Shift Keying
DVB-RCS	Digital Video Broadcast – Return Channel via Satellite
CSE	Creonic Simulation Environment
VHSIC	Very High Speed Integrated Circuits
VHDL	VHSIC Hardware Description Language
ASK	Amplitude Shift Keying
FSK	Frequency Shift Keying
PSK	Phase Shift Keying
BPSK	Binary PSK
QAM	Quadrature Amplitude Modulation
CRC	Cyclic Redundancy Check
FEC	Forward Error Correction
LDPC	Low-Density-Parity-Check
ARQ	Automatic Repeat Request
SISO	Soft In Soft Out
LLR	Log Likelihood Ratio
XML	Extensible Markup Language
AWGN	Additive White Gaussian Noise
FPGA	Field Programmable Gate Array
ASIC	Application Specific Integrated Circuit
ROM	read-only-memory
IP	Intellectual Properties

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The popularity of the wireless devices comes from several advantages related to this type of communication, i.e. mobility, easy installation and less cost for infrastructure. Hence it is vital to assure a reliable communication where errors can be autonomously fixed and information responsibly secured. The transmission over wireless channel results in frequency and phase offsets; additionally the received symbols are corrupted with noise. Therefore the estimation of the actual frequency and phase offset becomes a very critical task with high impact on communications performance; synchronization is a crucial part of each receiver in digital communication systems. In this context, throughout this work is proposed an implementation of a Fine Carrier Synchronization Unit that aims a better communication quality and lower its error rate.

Key-words: Wireless Communication, Synchronization, Frequency and Phase offsets.

Unidade de Sincronização Fina de Portadoras para Sistemas de Sincronização Turbo

RESUMO

A popularidade de equipamentos sem fio decorre de uma série de vantagens relacionadas a este tipo de comunicação, i.e. mobilidade, fácil instalação e menor custo para infra-estrutura. Consequentemente é vital garantir-se uma comunicação confiável onde erros podem ser automaticamente corrigidos e a informação segura. A transmissão sobre canais sem fio resulta em deslocamentos de frequência e fase; além disso, os símbolos recebidos podem ser corrompidos com ruído. Portanto, uma estimativa dos valores de deslocamento reais de frequência e fase se torna uma tarefa fundamental com grande impacto no desempenho da comunicação. Sincronização é uma parte crucial em cada receptor em sistemas de comunicação digital. Nesse contexto, ao longo deste trabalho é proposto a implementação de uma Unidade de Sincronização Fina de Portadoras que visa melhorar a qualidade da comunicação e diminuir a taxa de erros da mesma.

Palavras-Chave : Comunicação sem fio, Sincronização, Deslocamentos de Frequência e Fase.

1 INTRODUCTION

With the increase of mobility in our world, there is a rising necessity for communication and to have access to information, independently of the location of the individuals or information. Importance is given by the possibility that any phone call can be essential enough to save a life, close a business deal or provide hours of entertainment. Each of these examples of mobile communications proposes a challenge that can only be achieved with an efficient and reliable wireless communication.

Synchronization and channel coding/decoding are vital parts for wireless communication in every digital receiver– it decreases the errors and allows to reduce the transmission power respectively aiming for improvements in the performance [MENGALI, 1997]. With the intensification of devices using wireless data transmission technologies, it is required that there exist efficient and responsible ways to fix errors that may happen in this kind of transmission. When using wireless channel the received data will always be corrupted by some kind of noise – also timing, phase and frequency offset are introduced and somehow must be taken care of.

Receiver for wireless communication systems are in charge of the synchronization, decoding and detection [MEYR, 1997]. Detection is the ability to discern between information-bearing energy patterns and random energy patterns that distract from the information. Decoding is doing the opposite process of the encoding, in order to retrieve the original information.

In many coding systems, a decoder additionally produces soft (or side) information outputs to help another decoder identify and perhaps correct introduced errors. For example, in a Global System for Mobile (GSM) communication, an inner decoder comprising an equalizer generates a soft information output derived from path metric differences and an outer decoder comprising an error control decoder utilizes the output soft information to detect and correct introduced errors [REDL, 1995]. Soft information outputs have historically been generated by decoder in conjunction with the selection of the closest code word and its associated hard information output. Non algebraic decoders (e.g. Convolutional, Turbo) use also soft input information to increase decoding performance.

The reliability information comprising the soft information output is calculated for each individual symbol within the hard information output. The reliability of each symbol within the hard information output vector is derived without taking into consideration either the remaining symbols within that hard information output vector or any other considered code words. This is achieved by comparing the probability of the received data given a bit with a logical value of one was transmitted to the probability of the received data given a bit with a logical value of zero was transmitted.

At this moment, the Turbo codes are introduced with the purpose to improve the performance and the quality of the communication. That is only possible because a turbo code iteratively exchanges soft information, which will help the task of synchronization. Turbo codes are advanced codes which reduce bit-error-rate (BER) and frame-error-rate (FER) in comparison to other codes like convolutional codes or algebraic codes (e.g. Hamming and Reed-Solomon). Turbo Synchronization is a technique that uses the soft information to estimate the parameters needed to decrease the error rate of the communication [NOELS, 2003]. In turbo receivers, synchronization is a very challenging task – since they operate at very low signal-to-noise ratio (SNR) and therefore classical synchronizers may fail to provide reliable estimated parameters [LEHNIGK-EMDEN, 2008)]. Turbo decoders are working iteratively and after each iteration produce a soft output; it is important to note that the iterative nature of decoding allows to support synchronization.

Due to the time and purpose of this work, synchronization is the main subject. Channel decoding is left aside but it has potential to be a topic for future work. This work had as focus the frequency and phase synchronization of bursts with linear modulation, i.e. Quadrature Phase Shift Keying (QPSK) modulation. The system aims the Digital Video Broadcast – Return Channel via Satellite (DVB-RCS) standard, which is an ETSI satellite communication standard [ETSI, 2012].

Timing synchronization, sampling rate and other problems related to the communication are properly carried out before, which means that this work will focus only on the frequency and phase offset only.

It is a known fact that simulations can reduce development time and costs. A project was created at the Microelectronic Systems Design Research Group from the Technical University of Kaiserslautern, which developed the software Creonic Simulation Environment – CSE [MSDRG, 2012]. The purpose of CSE is to allow for the integration of complex simulations environments; there can be seen two distinguished models that compound CSE: the transmitter part and the receiving part. The simulation of the synchronization task for such communication systems has enormous importance on the whole project.

CSE was the starting point for the work developed. New features aiming the Fine Carrier Synchronization Unit were developed, tested and introduced into the already existing simulation environment – achieving a more powerful and wider software. Hardware implementations on Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) of these new features were also developed. Exactly the same functionality was intended in order to allow a complex and reliable comparison between both implementations. Therefore, it was possible to evaluate both – software and VHDL – according to the theory of communication systems and produce a good and complex statistical output.

1.1 Objectives of This Work

This work has as purpose two main objectives:

1. Implementation of the Fine Carrier Synchronization module in software and its integration to the CSE, as well as the implementation of the same module in VHDL.

2. Analysis, comparison and evaluation of the accuracy of the implemented software and its correspondent in VHDL.

1.1.1 Implementation of New Modules in Software

To achieve the first goal, the whole functionality of techniques of Synchronization for digital receivers was taken into account. Once the behavior was deeply comprehended, it became certain that a better and effective approach on the implementation of new synchronizations modules for any communications system was going to be reached. With the support of the CSE, it was possible to focus exactly on what this work proposes: the Fine Carrier Synchronization Unit. The software implementation of this unit and the integration with the CSE as a new module was aimed. An additional module for the transmitting part of the CSE (called Add Offset Module) was developed, which introduces the error of frequency and phase offset in the simulation system. This frequency and phase error will be estimated in the receiving part. The implementation of a correction module for phase and frequency offset is also necessary. To conclude, several upgrades on the Statistics Module of CSE were done in order to adapt it to the whole new set of functionalities performed by the CSE. With all the extensions of the CSE, it is possible to evaluate the statistical behavior (communication performance) of the new Fine Carrier Synchronization Unit.

1.1.2 Implementation of New Modules in VHDL

The hardware implementation in VHDL can be defined exclusively based on the needs of this work; it is not – so far – part of a bigger project. Xilinx ISE Design Suite V13.2 is the used digital system design tool and VHDL is the chosen language. It is a powerful and versatile description language, with multiple mechanisms to support design hierarchy and multiple levels of abstraction. The module in VHDL was implemented aiming exactly the same functionality of the Fine Carrier Synchronization Module developed in the software. In order to achieve this goal it was necessary to make a deep analysis of how to accomplish in VHDL several functionalities easily reached in the software – and to find the perfect approach during this “translation” from software to hardware. The VHDL was simulated and synthetized with the same framework that was used to its development.

1.1.3 Comparison between Software and VHDL

After the first objective of this work was successfully accomplished, the accuracy needed to be verified; it is essential that the output software and hardware are of a high excellence and properly studied. There is completely no point in developing, spending time and effort to analyze a system that does not fulfill the requirements of modern communication systems. It was also vital to re-examine new ways to improve processes and run them repeatedly - ensuring credibility, quality and functionality.

Hardware simulations and analysis are known for being extremely time-consuming. Taking into account the fact that the evaluation and the testing are extremely necessary in order to have a consistent and reasonable implementation; it was really important to find a way to bypass this problem and prove its functionality and reliability accordingly to the desired accuracy. Therefore, the idea to improve and optimize the simulations and analysis was to have the software to support this task. The hardware and software were implemented based on the same study; consequently they both do the exact same calculations and produce the same output – designated to different platforms. This way, it was intended to have a higher number of test cases on the software than on hardware but proceed with both evaluations together – also based on comparisons and exchange of information between the two implementations.

In order to achieve a deep analysis of the developed software and hardware – regarding the originals modules included in CSE – some features will be added to the Statistics Module with the purpose of statistically analyze the Fine Carrier Synchronization Module and its functionality.

Among the objectives of this work is the implementation of such Fine Carrier Synchronization Unit in VHDL. At this point it is crucial to understand some differences between the software and the hardware implementation, for example, there is completely no use to do the VHDL implementation of the Add Offset Module, which is part of the transmit model of CSE. The "real world" is in charge of this task – adding some frequency and/or phase offset to the set of bits. By developing exactly the same unit as in the software, it is possible to assure that it will have the same functionality. Therefore, all efforts must be done on the hardware implementation of the Fine Carrier Synchronization Module. Simulations and comparisons between both, the software and the hardware implementations, was part of the usual day-to-day work while this project was under development.

1.2 Structure of the work

The rest of this work is structured as follows. In chapter 2, an overview of the basic concepts needed and technical concepts involved on this work. Chapter 3 shows the whole functionality and architecture of CSE and advantages conquered by taking it as the first step into the development and implementation of the Fine Carrier Synchronization Unit. Chapter 4 deals with the software and hardware implementation of the new modules, specifically which were the adopted design options and the mathematical approach to every new module. In chapter 5, it is discussed how the validation was done, explaining the test environments, the scenarios, the methodology and the statistical evaluation of the developed software and VHDL. Chapter 6 brings the conclusion of this work. Finally, this work contains an annex A, that is the project of this work, an annex B, which is the full Xilinx ISE Design Tool Synthesis report, and an annex C, which is a summary of the thesis translated to Portuguese.

2 BASIC CONCEPTS

At this point, it is important to understand how the information can be transmitted from one point to another. By varying one physical property, i.e. voltage or current, the information can be transmitted in wires. By representing the value of this voltage or current as a function of time, (t), it is possible to model the behavior of the signal and analyze it mathematically.

In the XIX century, Jean-Baptiste Joseph Fourier, a French mathematician and physicist, proved that any periodic function, g(t), with period T can be constructed by the sum of a number (possibly infinite) of sines and cosines [OPPENHEIM, 1989].

$$g(t) = a_0 + \sum_{n=1}^{\infty} \left(a_n \cos\frac{n\pi t}{T} + b_n \sin\frac{n\pi t}{T} \right) \qquad (2.1)$$

Where a_n and b_n are the amplitude of the sine and cosine from the n harmonic terms and a_0 is a constant – this decomposition is called Fourier series. From this Fourier series, the function can be reconstructed; which means that, if the period T is known and the amplitudes also given, the original function of time can be found by making the sum of the equation 2.1. A data signal with a finite duration can be treated based on the premise that it repeats the same pattern; the interval between 0 and T is equal as the one between T and 2T – and it follows this configuration towards ∞.

Communication systems can be classified into two groups depending on the range of frequencies they use to transmit information. These communication systems are classified into baseband or pass band system. Baseband transmission sends the information signal as it is without modulation (without frequency shifting) while pass band transmission shifts the signal to be transmitted in frequency to a higher frequency and then transmits it, where at the receiver the signal is shifted back to its original frequency.

There are several different ways to accomplish the transmission, each one of them has their own bandwidth (difference between the upper and lower frequency – measured in Hertz), delay, cost and ease of installation and maintenance. They can be classified into guided media (i.e. copper wire and optic fiber) or not guided media (i.e. radio waves and laser).

Unfortunately media are not perfect, therefore the received signal is not the same as the transmitted signal – this difference can lead to error. Transmission lines suffer from three major problems: attenuation, delay distortion and noise.

These errors are responsible for wrong interpretation of the information transmitted, therefore is important to be prepared to deal with errors automatically. At this point the Synchronization of the signal fits to the whole system, because it tries to estimate and correct all the negative parameters – eliminating the negative effects – before the signal is interpreted.

2.1 Modulation

Computers are able to manage only binary numbers; it is possible to understand 0's and 1's, consequently it is necessary to develop some kind of alphabet or pattern to enable a proper and, many times, really complex communication with just two types of signal. The amplitude, frequency and phase of this signal can be modulated in order to transmit information. The amplitude modulation is characterized by the variation of two different amplitudes used to represent 0's and 1's. The frequency modulation, also known by frequency shift keying, are used two or more different tones. The phase modulation of the carrier is shifted systematically 0 or 180 degrees in equal periods of time. It is important to mention that this work will be restricted to digital modulation [TREES, 2001].

Modulation means the possibility to change one of the signal's properties (amplitude, frequency or phase) in such a way that different states are possible. Each possible state represents one symbol of the alphabet and it works with this "language" – the alphabet must be previously defined and known by both sides. Therefore, modulation means adding information on amplitude, frequency or phase. Figure 1 shows different modulations and how they can be represented and characterized to transmit a digital signal. In amplitude modulation or amplitude-shift-keying (ASK), two different amplitudes are used to represent 0 and 1. In frequency modulation or frequency-shift-keying (FSK), two – or more – different tones are used. In phase modulation or phase-shift-keying (PSK), the carrier wave is shifted 0 or 180 degrees at uniformly spaced intervals – the simplest form of PSK called Binary PSK (BPSK). The term keying is also widely used in the industry as a synonym for modulation.

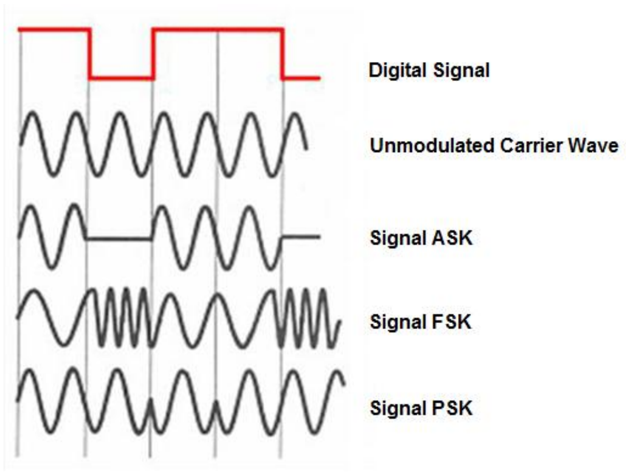


Figure 2.1: Different modulations applied to a digital signal BPSK

The communication can be represented in a complex plane and, for example, a Quadrature Phase Shift Keying (QPSK) has a defined alphabet with 4 symbols (00, 01, 10 and 11), equally divided on the plane. Consequently, it's natural to understand that for every 90° or π/2, that there will be the area where one symbol will be represented. It is usual to refer this organization of symbols in a complex plane as a constellation diagram – Figure 2.1. There are several modulations available and in use nowadays, i.e. BPSK, QPSK, 8PSK, 16-Quadrature Amplitude Modulation (QAM), 64-QAM, etc.

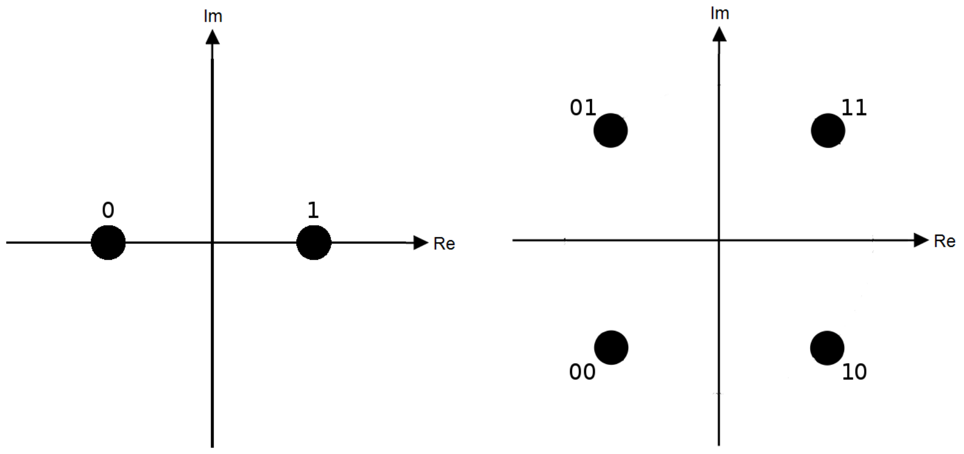


Figure 2.2: Constellation diagram for BPSK (left) and QPSK (right) with gray coding

In order to explain the figure 2.2, it is important to remind that the number of symbols (constellation point) per bits respects the formula 2.2, which represents that, for example, for every 1 bit, 2 symbols can be represented and for 2 bits, 4 symbols can be represented – accordingly to equation 2.2. This means that according to the modulation scheme a symbol represents 1, 2, 4 or more bits.

The modulation would work perfectly if there were no errors, noise and degradation of signal. To understand how an error occurs, it is important to note that once something went wrong, these symbols are not going to be on the exact expected place. Due to the noise, the symbol – the "point" in figure 2.2 – shifts its position in the complex plane when compared to the original position. This shifting will not always result in wrong interpretation – it can vary certain acceptable range and it will still be considered as the right symbol.

$$Number\_of\_Bits = log_2\, Number\_of\_Symbols \tag{2.2}$$

2.2 Signal Degradation by Noise

The noise is a random fluctuation in an electrical signal, which can arise in various forms. The amount of noise present is measured by the ratio of the signal power to the noise power, called signal-to-noise ratio (SNR). The signal power is denoted by S and the noise power by N, the signal-to-noise ratio is S/N. Usually, the ratio itself is not represented; instead, the quantity $10log_{10}S/N$ is given. These units are called decibels (dB).

Attenuation is the loss of energy as the signal propagates outward. The loss is expressed in decibels per kilometer. The quantity of energy lost depends on the frequency. To understand how the frequency affects this signal, imagine that it is not as simple wave form but as a component of a Fourier series; which each component has a different frequency and amplitude and is attenuated in a different proportion, resulting in different Fourier spectrum on the receiver's side.

To make things even worse, the several components of Fourier also have different propagation speed in the wire; this speed difference also leads to distortion of the signal received in the other end.

Another problem is the noise, which consists of an undesirable energy from other sources, not the transmitter. The thermal noise exists due to the random motion of electrons in the wire, which is inevitable. Crosstalk happens due to the inductive coupling between two wires that are close to each other. There is also the impulse noise, which happens due to spikes on the power line or other causes. For digital data, impulse noise can result in loss of one or more bits.

Of course a little variation on signal at the receivers side of the transmission is expected, the real problem starts when this error is bigger than the range of acceptable variation. In a very noisy environment, for matter of explanation, figure 2.3 makes it more understandable.

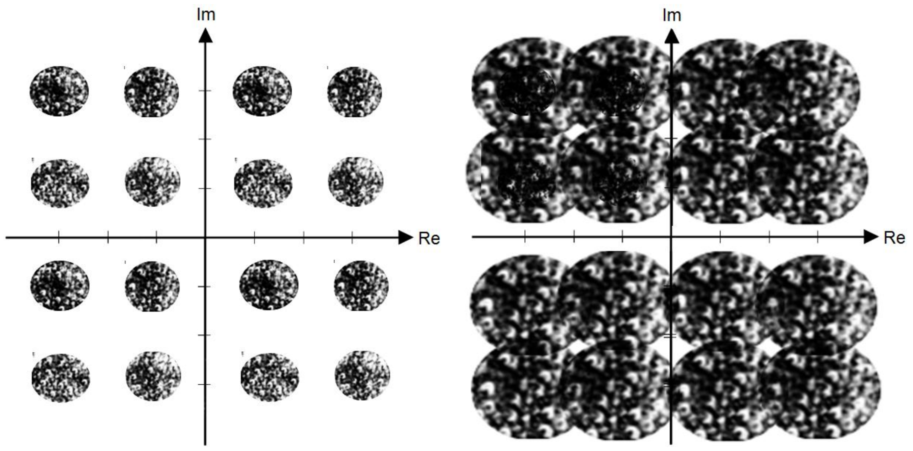


Figure 2.3: 16-QAM with an acceptable range variation (left – high SNR) and a very noisy range variation (right – low SNR)

Figure 2.3 comes with the purpose to illustrate and to remind that there might be different levels of noise variation; therefore, the task of classification of the received symbols may be affected or not due to the noise of the system – it is possible that some noise is harmless to the communication quality. On a high SNR environment the symbols do not have massive losses caused by noise (left part of figure 2.3); it is possible that the variation may not be enough to disturb and cause errors on the interpretation of the symbols. With a low SNR (right part of figure 2.3), the analysis and classification of the symbols can be disrupted since the range of variation of different symbols have an intersection where two different interpretations for the same coordinate is possible – which results in an error of reading the symbol. It is also important to remind that figure 2.3 does not represent an instant “picture” of the communication; it symbolizes all the possible states of the media.

SNR is going to define how bad the signal will get after going through the channel. It is directly influenced by the noise variation of the channel. Also included in the negative parameters of any synchronization are the frequency and phase offset. Figure 2.3 can be observed also by differentiating the values of SNR: left part of figure 2.3 is the representation of a high SNR value; while the right part of figure 2.3 corresponds to a low SNR value.

The bit error rate (BER), in a digital transmission, is the percentage of bits with errors divided by the total number of bits that have been transmitted, received or processed over a given time period. The rate is typically expressed as 10 to the negative power. For example, four erroneous bits out of 100,000 bits transmitted would be expressed as $4 \, x \, 10^{-5}$, or the expression $3 \, x \, 10^{-6}$ would indicate that three bits were in error out of the 1,000,000 transmitted. BER is the digital equivalent of signal-to-noise ratio in an analog system.

As it can be seen in figure 2.4, the relation between BER and SNR in a simulation of a communication using QPSK shows that with the increase of the SNR, the BER will get lower (important to remind that it will never be errorless). The BER behavior of the transmitted signal is the reason for using coding techniques [TAUB, 2008].

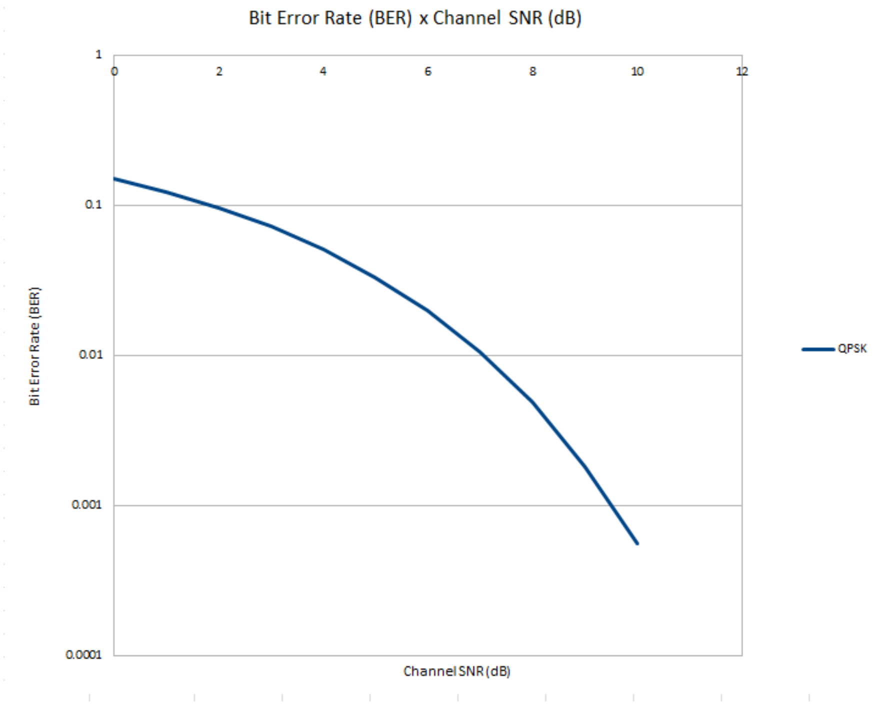


Figure 2.4: Bit Error Rate (BER) x Channel SNR (in dB) in a QPSK communication

Frequency offset exists as the consequence from the difference between the oscillator from the transmitter (TX) and the one from the receiver (RX); the oscillators from TX and RX cannot be exactly equal. To comprehend how the frequency offset can be observed, it is primordial to understand that they do not come at the exact same moment; it is an error that increments its effect while the transmission proceeds.

For example, figure 2.5 represents the transmission of “10101010” with a frequency offset f, which is responsible for adding a constant and incrementally error to every symbol. This is the reason why the symbols have a variation from the place identified on the figure as “first” and the “last”. Every upcoming symbol will be a bit more displaced regarding the original position of the “10” on the complex plane. Due to didactic purposes, the whole bit stream is represented on a complex plane as an accumulator – if an instantaneous picture were taken of the complex plane, just one “10” symbol would be present at a moment and this would not be helpful for the comprehension of the concepts [BRACK, 2005b].

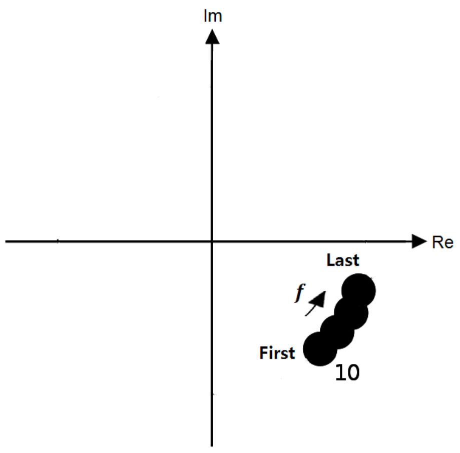


Figure 2.5: Illustration of a frequency offset (f) in a QPSK modulation

Frequency offset have a higher impact on symbols as they are more close to the end of the burst – the impact of the frequency offset will not be the same for every symbol, it is sequentially added to every new symbol [BRACK, 2005a]. On the other hand, the phase offset is the same to the whole bit stream – it acts in every symbol exactly in the same way. In figure 2.6 is explained how the phase offset works in a QPSK modulation.

The analysis and classification of each received symbol is a receiver's task, the interpretation of the symbol is qualified in a complex plane. Figure 2.6 a) shows a QPSK transmission where a symbol "10" was transmitted with a phase offset (α) – assuming that the frequency offset is zero. The classification of this symbol does not imply any error in its interpretation; since the QPSK implies that every symbol has a 90 degrees range, it is possible to have a phase offset with no error on the interpretation of the signal. Figure 2.6 a) is important to understand that not every phase offset would result in error.

On the other hand, with figure 2.6 b) it is prominent that the phase offset introduced will result in error of the interpretation of this symbol. The symbol was located in a quadrant of the complex plane and after the addition of the phase offset it is on a different one. This means that its interpretation will result in error, which can be corrected with the right estimation of this phase offset and its future correction [VITERBI, 1983].

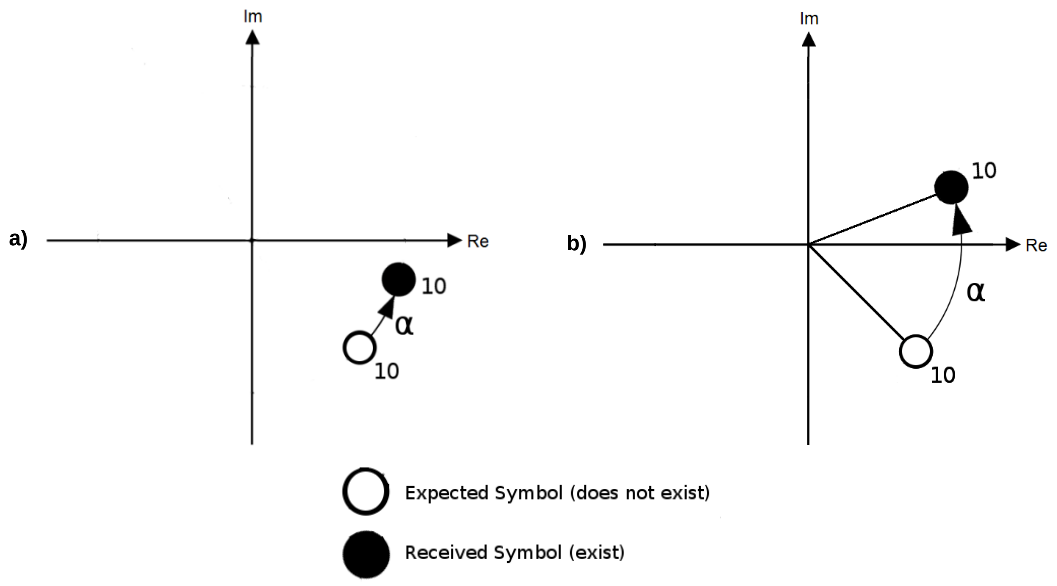


Figure 2.6: Illustration of phase offset (α) in a QPSK modulation

2.3 Channel Capacity

Besides the errors, it is mandatory to take into account physical limitations of the media. Henry Nyquist has developed studies not just related to channel capacity but also to the sampling rate and the comprehension of both is f undamental to proceed. Channel capacity is the maximum data transfer which can occur when using the communication channel with bandwidth B – channel capacity can be defined with equation 2.3 [TANENBAUM, 2003].

$$Channel\ Capacity_{max} = 2B$$

In 1924, Nyquist realized that even a perfect channel has a finite transmission capacity. He derived an equation expressing the maximum data rate for a finite bandwidth noiseless channel. In 1948, Claude Shannon carried Nyquist's work further and extended it to the case of a channel subject to random (thermodynamic) noise.

Nyquist proved that if an arbitrary signal with highest frequency component, in hertz, is f_{max} the sampling rate must be at least $2 * f_{max}$ or twice the highest analog frequency component. The sampling in an analog-to-digital converter is done by a pulse generator (clock). If the sampling rate is less then $2 * f_{max}$, some of the highest frequency components in the analog input signal will not be correctly represented in the digitized output. When such a digital signal is converted back to analog form by a digital-to-analog converter, false frequency components appear that were not in the original analog signal. This undesirable condition is a form of distortion called aliasing. In equation 2.3, the signal consists of V discrete levels.

$$Maximum\ data\ rate = 2f_{max} \log_2 V\ bits/\sec \qquad 2.3$$

Figure 2.7 shows that a failure on the sampling rate can determine a serious mistake on the interpretation; it is the transmission of a code word 010101, by not obeying the Nyquist theorem, the interpretation will be 000 (left). By taking a correct sampling rate the right information is correctly acquired: 010101(right).

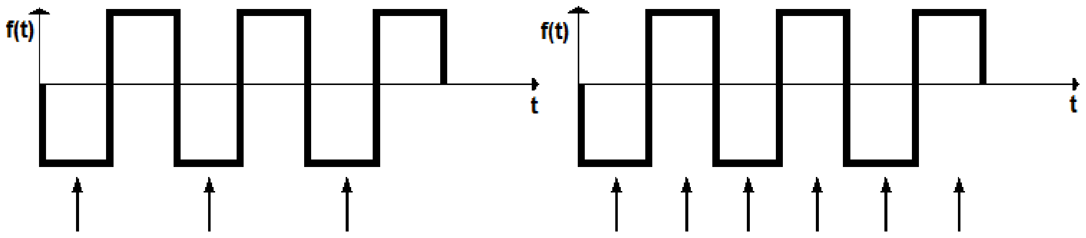


Figure 2.7: Illustration of a not properly sampling rate (left) and a properly sampling rate (right) accordingly to Nyquist theorem

Figure 2.7 represents the problem on the task of conversion of an analog-digital conversion. It is necessary to get a discrete number of samples in a continuous signal. The main problem is on the sampling/seconds rate that should be taken. A small number can result in a really poor representation of the signal. At first, the process of sampling of an analog signal can be thought that always will be losses of the information and that with the best sampling rate the losses are going to be the smaller possible; however, Shannon's theorem shows that this is not always true. Under certain conditions, the sampling of a signal can transmit precisely all the information contained in the signal. This means that the signal can be perfectly recovered from samplings without any decrease on the signal quality.

If random noise is present, the situation deteriorates quickly. Since there is always random (thermal) noise present due to the motion of the molecules in the system, this cannot be left aside. Shannon's major result is that the maximum data rate of a noisy channel whose bandwidth is f_{max}, in hertz, and whose signal-to-noise ratio is S/N, is given in equation 2.4. Shannon's result was derived from information-theory arguments and applies to any channel subject to thermal noise.

$$Maximum\ Number\ of\ bits/\sec = f_{max}\log_2\left(1+\frac{S}{N}\right) \qquad 2.4$$

2.4 Techniques of Detection and Correction of Errors

Among the error detecting methods there are Parity Check Method and Cyclic Redundancy Check (CRC) Method. Parity check method is the method where one parity check bit is used along with each character code to be transmitted; as a simple example of an error-detecting code, consider a code in which a single parity bit is appended to the data. The parity bit is chosen so that the number of 1-bits in the code word or character code to be transmitted or recorded is even or odd.

For example, when 10110101 is sent in even parity by adding a bit at the end, it becomes 101101011, whereas 10110001 become 101100010 with even parity. A code with a single parity but has a distance 2, since any single-bit error produces a code word with the wrong parity, it can be used to detect single errors. Two errors cannot be detected by this scheme as the total number of 1s in the code will remain even after two bits change. As the probability of more than one error occurring is in practice very small, this scheme is commonly accepted as sufficient.

Instead of appending a parity check, which makes the total of 1-bits in the code even, one may choose to append a parity check bit which makes the number of 1-bits in the code odd. Such parity check is known as an odd parity bit. This scheme also facilities detection of a single error in a code.

The CRC is wide spread for error detection, which codes are based upon treating bit steam as a representation of polynomials with co-efficient of zero and one only.

A $k + bit$ frame is regarded as the co-efficient list for a polynomial with k terms, ranging from $xk - 1$ to x0. Such a polynomial is said to be of degrees $k - 1$ the high order (left most) bit is the coefficient of $x\ k - 1$, the next bit the coefficient of $x\ k - 2$, and so on. I.E. 110001 has six bits and thus represents a six-term polynomial with coefficient 1, 1, 0, 0, 0 and 1: $x^5 + x^4 + x + 0$.

When the polynomial code method is employed, the sender and receiver must have agreed upon a generator polynomial, $G(x)$, in advance. Both the high-and-low-order bits of the generator must be 1. To compute the checksum for some frame with m bits, corresponding to the polynomial$M(x)$, the frame must be longer than the generator polynomial.

The idea is to append a checksum to the end of the frame in such a way that polynomial represent by the check summed frame, it tries dividing by $G(x)$. If there is a remainder, there has been a transmission error.

The correction of errors is more difficult than the detection. In error detection, it is just a system where it just looks to find if any error has occurred. The answer is simple yes or no. In error correction, it is demanded to know the exact number of bits that are corrupted and more importantly, their location in the message [CLARK, 1981]. The number of errors and the size of the message are important factors. Supposing the correction of one single error in an 8-bit data unit, it is necessary to consider eight possible error locations; if there were two errors in the same data unit, it is essential to consider 28 possibilities. Therefore, it is possible to imagine the receiver's difficulty in finding 10 errors in a data unit of 1000 bits.

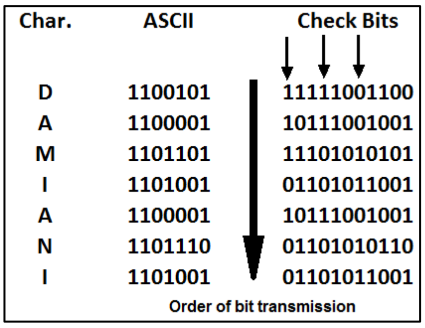
To understand how errors can be handled, it is necessary to look closely at what an error really is. Normally, a frame consists of m data (i.e., message) bits and r redundant, or check, bits. Let the total length be n (i.e., $n = m + r$). An n-bit unit containing data and check bits is often referred to as an n-bit code word.

Given two code words, say, 10001001 and 10110001, it is possible to determine how many corresponding bits differ. In this case, 3 bits differ. The number of bit position in which two code words differ is called the Hamming Distance (Hamming, 1950). Its significance is that if two code words are a Hamming distance d apart, it will require d single-bit errors to convert one into the other.

In most data transmission applications, all 2^m possible data messages are legal, but due to the way the check bits are computed, not all of the 2^n possible codewords are used. Given the algorithm for computing the check bits, it is possible to construct a complete list of the legal codewords, and from this list find the two code words whose Hamming distance is minimum. This distance is the Hamming distance of the complete code.

The error-detecting and error-correcting properties of a code depend on its Hamming distance. To detect d errors, you need a distance d+1 code because with such a code there is no way that d single-bit errors can change a valid code word into another valid code word. When the receiver sees an invalid code word, it can tell that a transmission error has occurred. Similarly, to correct d errors, you need a distance 2d+1 code because that way the legal code words are so far apart that even with d changes, the original code word is still closer than any other code word, so it can be uniquely determined.

There is a trick that can be used to permit Hamming codes to correct burst errors. Sequences of k consecutive code words are arranged as a matrix, one code word per row. Normally, the data would be transmitted one column at a time, from left to right, as shown in figure 2.8. To correct burst errors, the data should be transmitted one column at a time, starting with the leftmost column. When all k bits have been sent, the second column is send, and so on. When the frame arrives at the receiver, the matrix is reconstructed, one column at a time. If a burst error of length k occurs, at most 1 bit in each of the k code words will have been affected, but the Hamming code can correct one error per code word, so the entire block can be restored. This method uses kr check bit to make blocks of km data bits immune to a single burst error of length k or less. Where r represents the number of verification bits and m the number of bits in each message.



Char.	ASCII	Check Bits
D	1100101	11111001100
A	1100001	10111001001
M	1101101	11101010101
I	1101001	01101011001
A	1100001	10111001001
N	1101110	01101010110
I	1101001	01101011001

Order of bit transmission

Figure 2.8: Hamming code trick to correct burst errors

It is important to remind that several others forward error correction (FEC) codes are available now at the academic and industry field, for example Reed-Solomon (RS), Low-Density-Parity-Check (LDPC), etc [SNIFFIN, 2009].

Also important is to mention a method to recover from errors, Automatic Repeat Request (ARQ). This method is an error control for data transmission that makes use of error-detection codes, acknowledgment and/or negative acknowledgment messages, and timeouts to achieve reliable data transmission. An acknowledgment is a message sent by the receiver to indicate that it has correctly received a data. Important to keep in mind that ARQ is not a correction method (it does not correct any wrong information received) but a recovery method (it recovers the system from wrong information received).

Unusually, when the transmitter does not receive the acknowledgment before the timeout occurs (i.e., within a reasonable amount of time after sending the data frame), it retransmits the frame until it is either correctly received or the error persists beyond a predetermined number of retransmission.

ARQ is appropriate if the communication channel has varying or unknown capacity, such as is the case on the Internet. However, ARQ requires the availability of a back channel, results in possibly increased latency due to retransmissions, and requires the maintenance of buffers and timers for retransmissions, which in the case of network congestion can put a strain on the server and overall network capacity.

Turbo codes are a development in the field of forward-error-correction channel coding. Turbo codes make use of three simple ideas: parallel concatenation of codes to allow simples decoding; interleaving to provide better weight distribution; and soft decoding to enhance decoder decisions and maximize the gain from decoder interaction [ALLES, 2007].

A turbo code is formed from the parallel concatenation of two codes separated by an interleaver. Interleaving is a way to arrange data in a non-contiguous way to increase performance. A generic design of a turbo code is shown in figure 2.9.

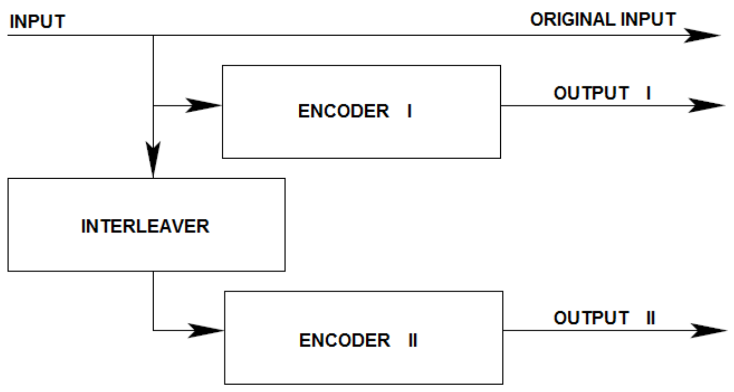


Figure 2.9: A generic turbo encoder

Although the general concept allows for free choice of the encoders and the interleaver, most designs follow the same ideas:

- The two encoders used are normally identical;
- The code is in a systematic form, i.e. the input bits also occur in the output;
- The interleaver reads the bits in a pseudo-random order.

This serves two purposes. Firstly, if the input to the second encoder is interleaved, its output is usually quite different from the output of the first encoder. This means that even if one of the output code words has low weight, the other usually does not, and there is a smaller chance of producing an output with very low weight. Higher weight is beneficial for the performance of the decoder. Secondly, since the code is a parallel concatenation of two codes, the divide-and-conquer strategy can be employed for decoding. If the input to the second decoder is scrambled, also its output will be different or “uncorrelated from the output of the first encoder. This means that the corresponding two decoders will gain more from information exchange.

In the traditional decoding approach, the demodulator makes a “hard” decision of the received symbol, and passes to the error control decoder a discrete value, either a 0 or a 1. The disadvantage of this approach is that while the value of some bits is determined with greater certainty than that of others, the decoder cannot make use of this information.

A soft-in-soft-out (SISO) decoder receives as input a “soft” (i.e. real) value of the signal. The decoder then outputs for each data an estimate expressing the probability that the transmitted data bit was equal to one. In the case of turbo codes, there are two decoders for outputs from both encoders. Both decoders provide estimates of the same set of data bits, but in different order. If all intermediate values in the decoding process are soft values, the decoders can gain greatly from exchanging information, after appropriate reordering of values. Information exchange can be iterated a number of times to enhance performance. At each round, decoders re-evaluate their estimates, using information from the other decoder, and only in the final stage will hard decisions be made, i.e. each bit is assigned the value 1 or 0. Such decoders, although more difficult to implement, are essential in the design of turbo codes.

2.5 General Considerations

By comprehending how it can be possible to transmit information over wires and wireless systems, and mainly its importance to modern communication systems; it is possible to understand why several studies on this topic must be realized - high quality systems are demanded throughout the most dynamic range of systems that depends on this kind of communication. The system must somehow certify that the way of sending information through a channel works and is efficient (modulation), the same system must be aware of interferences and it is also important to be able to work independently; detecting and correcting possible errors in an autonomous way.

As a way of increasing the quality and the utilization of time and resources when developing such systems, a simulation environment must be used. In order to enable tests, prototyping, education and the statistical analysis of such systems is mandatory. The next chapter is dedicated to explain which simulation environment was used during this work. This system – Creonic Simulation Environment – is introduced; its architecture and design options are presented and discussed, as well as advantages and disadvantages.

3 CREONIC SIMULATION ENVIRONMENT - CSE

CSE was developed by the Microelectronic System Design Research Group and Dr.-Ing Timo Lehnigk-Emden and Dr.-Ing Matthias Alles – both are now former researchers from the Microelectronic System Design Research Group. CSE was one of the successful academic projects that had become a company in real world – Creonic IP Cores & System Solutions GmbH [CREONIC, 2012]. Creonic is a spin-off of the University of Kaiserslautern, founded in 2010. With broad technical knowledge and experience of business activities, the founding of Creonic was the logical consequence after successful academic projects.

The main application of the Microelectronic System Design Research Group is decoding. The purpose of CSE is to have a simulation environment for decoders – there are similar tools commercially available from Synopsys and Cadence. These commercial tools provide more features but also require money for licensing and normally the use is more complex than CSE.

The development of communication systems is a very dynamic field with respect to technical progress. In this context, Creonic collaborates closely with the Microelectronic System Design Research Group, which has an experience in the field of communication. This way it is possible to stay up-to-date regarding the state-of-the-art of science and technology – the results are the highest performance and at the same time low energy consumption.

Projects with such delicate design and development questions must be highly detail oriented. Hence, choices of design have been adopted and respected throughout the whole development of the software. For example, fixed interface and configuration procedures for functional modules, strict coding and documentation guidelines, and fully object oriented design were defined. For the documentation purposes, a documentation tool called Doxygen [DOXIGEN, 2012] was used – it generates the documentation from a set of documented source files automatically.

All the CSE project choices were done according to the goal of developing a system which can provide reduction of the development time and costs, facilitate the use, reusability and extensibility to new applications or standards.

By defining smart choices in the beginning of the project, i.e. reusability, the benefits to the project can be seen during the whole development. Reused software, which has been tried and tested in working systems, are more intend not to reveal any design and implementation fault. If software exists, there is less uncertainty in the costs of reusing that software than in the costs of development – it reduces the margin of error in project cost estimation.

CSE environment consists mainly of the library directory; which contains all the classes of the CSE, and the chains directory; containing some simulation chains, which use the library. Requirements to run the CSE are a g++ version 4.x under GNU/Linux, MS Windows or MacOS X, Microsoft Visual C++ under MS Windows and CMake – tool designed to build, test and package software.

C++ was the programming language chosen to the implementation of CSE. Which is a clever option based on the design choices made by the original developers of the software. C++ contains a good and rich standard library, compared to C, and also supports both the structured programming and object orientation [DALE, 2004].

3.1 CSE Architecture

To completely comprehend the architecture of a complex simulation environment like CSE, it is necessary to come up with a specific approach in order not to miss any information or leave room for misunderstanding. On section 3.1.1 the class structure of any module and how the infrastructure provided by the system helps all the modules will be presented. On section 3.1.2 the original CSE modules are presented – they are all functional modules ready to enable the user to create new simulation chains.

3.1.1 CSE Modules and Configuration

This simulation environment is extremely useful for projects related to communication systems; the simulations costs – related to configuration and connection – that once used to cost a lot of time, hours or days for the developers, now can be easily finished in a few minutes.

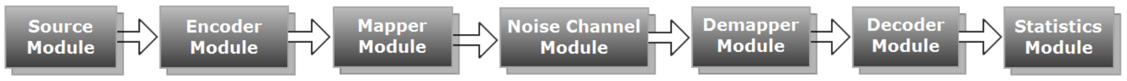


Figure 3.1: Original simulation chain and functional modules connection available on CSE

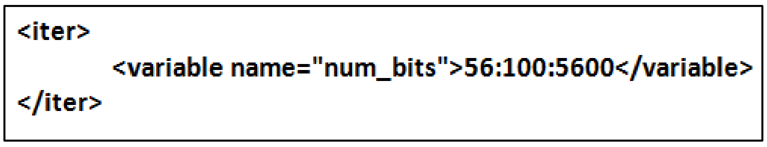
Figure 3.1 shows the simulation chain available on CSE, it represents the whole path that the bit stream will make. The source module generates random bit sequences. It is responsible for calculating one block of bits and storing them into an output buffer. The Encoder module changes the block of data bits (information bits) into a code; the code may be optimized for purposes of compressing for transmission or storage, encrypting or adding redundancies to the input code. The Mapper module is responsible for mapping bits to modulation symbols. The Noise Channel module adds to the simulation effects of real life – the impairment to communication is a linear addition of white noise with a constant spectral density and a Gaussian distribution of amplitude. The Demapper module receives the symbols from the channel and extracts the hard bits, the log likelihood ratio (LLR) values (LLR – is a statistical test used to compare the fit of two models) and the bit probabilities. The Decoder module will be in charge of the reverse operation of the encoder – changing the code into a set of signals and doing the estimation of the information bits. The Statistics module is responsible for comparing the input bits and the output bits, it takes into account the total amount of bits and how many of them are different – in other words, it is to verify how “faithful” is the output after passing through the entire simulation chain. Besides it computes this information into different statistical parameters.

The model used inside the Noise Channel Module to generate the noise is the Additive White Gaussian Noise – AWGN. Additive White Gaussian Noise (AWGN) is a white noise that is distributed according to the Gaussian distribution curve and a constant spectral noise power over the channel bandwidth. The AWGN noise affects the useful signal, which is linearly superimposed, hence the term additive. The total signal at the channel end corresponds to the addition of the amplitude of the input information signal and the amplitude of the noise signal.

All the configurations of the simulation using CSE are defined in the Extensible Markup Language (XML) configuration file, on this file it is possible – and necessary – to define all the configurations and parameters to every module instantiate. The XML file is used during runtime execution; therefore, it is not necessary to build the library if the aim is to build a simulation chain [RAY, 2001].

A typical XML configuration file starts with the main tag <cse_chain> that contains all other tags. Each module is identified with the <instance_name> tag, which corresponds to the <instance_name> parameter that each module contains. By default, the instance name corresponds to the class name of the module. An optional <global> tag is also available. The purpose of this tag is to spread values to a multiple number of modules, without the need to set this value at multiple places of the XML – similar to the definition of "#define" in C++. Additional tags are dependent on the Parameter class of the module.

Up to now all the configuration works in a static environment. Therefore, the <iter> tag is introduced in the global section in order to define variables which will create a static configuration for each value. In case multiple iteration variables exist, the order of iteration variable definition is important – it turns several simulations into one with just one tag, simplifying a lot the work that has to be done. For example, in figure 3.2 by defining <num_bits> inside the <iter> tag, it means that it will vary from 56 to 5600 with a pace of 100.



```
<iter>
        <variable name="num_bits">56:100:5600</variable>
</iter>
```

Figure 3.2: Explanation of the <iter> tag

An extra <param_unit> tag exists for configuration of parameter units. The <unique_id> attribute allows an automatic instantiation of the parameter unit during runtime. Parameter units allow the translation of parameters into other parameters.

Figure 3.3 shows an excerpt of the typical XML file with global parameters definition, parameters definition and the instantiation of three modules and its parameters: Mapper, Channel_AWGN and Demapper.

```
<global>
    <variable name="info_bits">488</variable>
    <variable name="other_bits">488</variable>
    <variable name="mapping">MAP_QPSK</variable>
    <variable name="num_bits_per_symbol">2</variable>
    <variable name="noise_variance"><param param_unit="SNR" name="noise_variance"/></variable>
    <iter>
      <variable name="es_n0">1.8:0.2:2.2</variable>
    </iter>
  </global>

  <param_unit unique_id="Param_Unit_SNR">
    <instance_name>SNR</instance_name>
    <input_snr><global_variable name="es_n0"/></input_snr>
    <input_type>ES_N0</input_type>
    <num_info_bits><global_variable name="info_bits"/></num_info_bits>
    <num_other_bits><global_variable name="other_bits"/></num_other_bits>

    <param name = "noise_variance"></param>
    <param name = "eb_n0"></param>
    <param name = "es_n0"></param>
  </param_unit>

  <module>
    <instance_name>Mapper</instance_name>
    <mapping><global_variable name="mapping"/></mapping>
  </module>

  <module>
    <instance_name>Channel_AWGN</instance_name>
    <noise_variance><global_variable name="noise_variance"/></noise_variance>
  </module>

  <module>
    <instance_name>Demapper</instance_name>
    <mapping><global_variable name="mapping"/></mapping>
    <noise_variance><global_variable name="noise_variance"/></noise_variance>
  </module>
```

Figure 3.3: Excerpt of the XML configuration file

3.1.2 CSE Output

The CSE output is organized in a XML file, which brings the information of how the simulation chain was built as well as what parameters were used to perform the simulation. Besides this, it brings the statistics module output – which is the most important part at this moment. In figure 3.4 an excerpt of the simulation configuration details present on every XML output file is shown. The actual statistical output values are presented on figure 3.5, i.e., error rate bits, error rate blocks, etc.

All the data created and that goes through the simulation chain is analyzed by the statistics module at the end of the simulation – it gathers information at all moments but just analyzes in the end of the whole set of bursts since it takes into account not just one burst to get a result of what has been going on with the system. This is possible due to the fact that it is connected to both, the beginning and the end of the simulation chain.

```xml
<configuration>
    <cse_chain>
        <global>
            <variable name="mapping">MAP_QPSK</variable>
        </global>
        <module>
            <instance_name>Mapper</instance_name>
            <mapping>MAP_QPSK</mapping>
        </module>
        <module>
            <instance_name>Channel_AWGN</instance_name>
            <noise_variance>0.251188643</noise_variance>
            <start_seed>846546818653215</start_seed>
        </module>
        <module>
    ...
    </cse_chain>
</configuration>
```

Figure 3.4: Excerpt of the input configuration copied to the XML output file

```xml
<result>
    <module name="Statistics_Error_Rates">
        <status_out name="num_total_bits">
            <value index="0">1120000</value>
        </status_out>
        <status_out name="num_diff_bits">
            <value index="0">127145</value>
        </status_out>
        <status_out name="error_rate_bits">
            <value index="0">0.113522</value>
        </status_out>
        <status_out name="num_total_blocks">
            <value index="0">10000</value>
        </status_out>
    ...
    </module >
</result>
```

Figure 3.5: Excerpt of the statistical results printed on the XML output file

3.1.3 CSE Class Structure

The simulation environment is composed of functional modules; they work as versatile pieces available to be organized together following the needs of the user, i.e. noise generator or channel decoder. These functional modules are connected with each other providing the possibility to create complex simulation chains. In order to have a uniform and constant development of any module, there are implementation concepts related to these functional modules that must be respected and preserved. Each module is derived from two or three classes that have to be written by the module designer:

- The interface class, which defines the interface for the input and output of data to process. Furthermore, it provides status information, like number of used decoder iterations.

- The parameter class, which defines the parameters for configuration of the module functionality (e.g., chosen algorithm).
- The share class, which defines functionality that can be shared among multiple modules – this class is optional.

The interface and parameter classes have to be derived from their base classes, which are provided by the CSE infrastructure. The base classes (Base_Interface and Base_Parameter) are parent of all interfaces and parameters, and therefore of all modules of the simulation chain. These classes provide higher level data structures, such as lists containing all the parameters or data ports of a functional module. Figure 3.6 illustrates the functional module concept (relation between classes and how they are organized) – it represents the general implementation concepts like derivation structure of functional modules. All the classes defined with the subtitle "(infrastructure)" are provided by the CSE system and offer to all the system basic facilities, services and facilities to the organization of the whole system itself, i.e., list of all input/output ports and list of all parameters of each module. All the classes with the subtitle "(user)" are the ones that the user can create (or just use if the modules are already on the original CSE, i.e., mapper, demapper, encoder, decoder, channel_awgn).

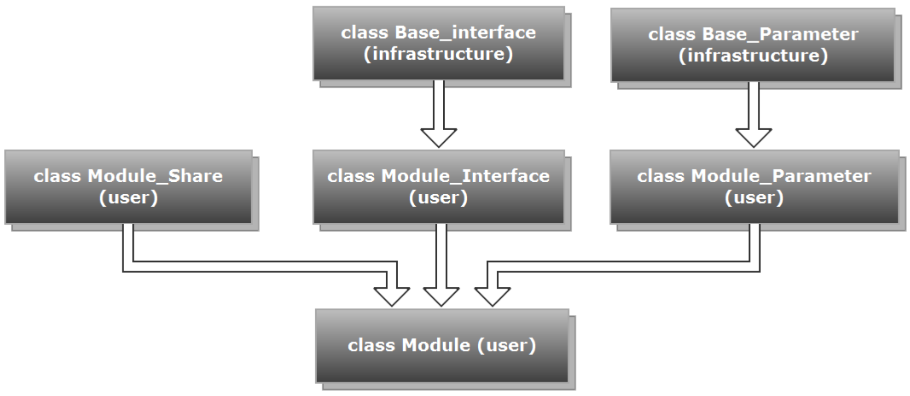
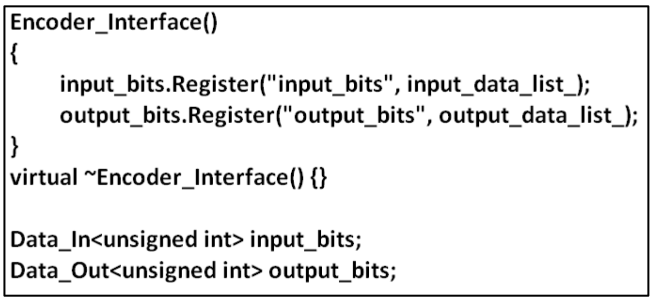

class Base_interface (infrastructure)
class Base_Parameter (infrastructure)
class Module_Share (user)
class Module_Interface (user)
class Module_Parameter (user)
class Module (user)

Figure 3.6: Organization of the module classes

To a better comprehension on how the classes work together, an example will follow. A channel encoder gets bits to encode and has as output the corresponding code word. These two data interfaces are defined on the interface class of the encoder – as can be seen on figure 3.7.



```
Encoder_Interface()
{
      input_bits.Register("input_bits", input_data_list_);
      output_bits.Register("output_bits", output_data_list_);
}
virtual ~Encoder_Interface() {}

Data_In<unsigned int> input_bits;
Data_Out<unsigned int> output_bits;
```

Figure 3.7: Excerpt of the Encoder_Interface file – creation and definition of the interfaces

The communication between a module and its environment is called "port" – the data type for a port is one of the two template classes available within the CSE library:

Data_In or Data_Out. Data_In is the input port class of the modules, which will enable the possibility to, when instantiating it, define how are the input types and it will also be responsible to create and instantiate the buffer to connect with the input port. The data are stored in a Buffer class instance.

The interface classes can be shared easily for a group of similar modules like all encoders, because a wide range of encoders have the same data I/O. The name of the interface class is the class name of the module with the "_interface" suffix, as "encoder_interface" shown in figure 3.7.

The parameter class defines all parameters that can be changed during run-time to adapt the module functionality e.g., defining the number of symbols or defining which is the modulation to be used. A parameter is defined to be of type available in the CSE library: Param. The parameter class is usually very module specific, because the parameters have a low similarity for different modules. The name of the parameter class is the class name of the module with the "_parameter" suffix. The reason to split the parameter in an extra is that all aspects like definition, default values, and automatic configuration over XML of the parameters are concentrated in a single class in one file.

The Share class contains functionality that can be shared among multiple modules. E.g., LDPC decoders can share the check node functionality or mapper and demapper use the same constellation points for a given modulation. By putting such functions into the share class, a high reuse can be achieved. In general, the parameters of the encoder, as an example, are shown in figure 3.8 – there will be always a value classified as "default", preventing the case when the user does not define what the parameter values to use are.

```
/**************
* Parameters *
**************/
enum LDPC_CODE {
      UMIC_P372_N14880_R050,
      UMIC_P372_N14880_R075,
      UMIC_P372_N14880_R080
};

Param<LDPC_CODE> ldpc_code;

/******************
* Default values *
******************/
void Set_Default_Values()
{
      ldpc_code.Init(UMIC_P372_N14880_R050, "ldpc_code", param_list_);
      ldpc_code.Link_Value_String(UMIC_P372_N14880_R050, "UMIC_P372_N14880_R050");
      ldpc_code.Link_Value_String(UMIC_P372_N14880_R075, "UMIC_P372_N14880_R075");
      ldpc_code.Link_Value_String(UMIC_P372_N14880_R080, "UMIC_P372_N14880_R080");
}
```

Figure 3.8: Excerpt of the Encoder_Parameter file – creation and definition of the default parameters values

Configuration of the functional module is a challenging task. For a standard-compliant simulation environment many modules are connected via their data interface. In order to perform the correct operation, it is mandatory to configure every single module. Usually a standard provides a header word, defining the current operation of the circuit. This header word has to be translated into other parameters, such as modulation scheme, coding scheme, polynomials, etc.

3.2 Development of New Modules

The CSE goal is to allow the user to create its own simulation environment with a rich documentation. It encourages the users to create new simulation chains and also new functional modules as well as test them inside the whole system – analyzing the results and the performance of the new changes or applications. New simulation chains can be made using the existing modules already provided by the CSE (reorganizing or/and changing parameters), creating a completely new simulation with brand new modules or using new modules with the original CSE modules.

For writing new modules for the simulation there are some steps that must be done. In section 3.2.1 the instructions on how to create a parameter class for the module will be presented. Section 3.2.2. will bring the instructions on choose or write an interface class. Finally, section 3.2.3. defines the instructions to write the module itself. In these sections all the steps are explained in more detail based at the example of a hardware-compliant LDPC decoder model.

3.2.1 Create a Parameter Class for the Module

As a first step, it is necessary to generate the parameter class that corresponds to the module intended to implement. This parameter class contains all parameters that are needed by the decoder to run. In this example there are things like quantization, scheduling, number of iteration, etc. The parameter class – an example is shown in figure 3.9 – is derived from the class Base_Parameter. The include's directives have to contain relative paths to any directory for the base and the assistance directories.

```
#ifndef DEC_LDPC_BINARY_HW_PARAM_H_
#define DEC_LDPC_BINARY_HW_PARAM_H_

#include "../base/base_param.h"

class Decoder_LDPC_Binary_HW_Parameter : public Base_Parameter
{
public:
            Decoder_LDPC_Binary_HW_Parameter()
            {
                  instance_name(Unique_ID());
                  Set_Default_Values();
            }
            virtual ~Decoder_LDPC_Binary_HW_Parameter() { }
            static string Unique_ID(){ return "Decoder_LDPC_Binary_HW";}
protected:
};
#endif
```

Figure 3.9: Code of an empty parameter class for a LDPC decoder

Parameters for the modules are added to the public part of the class and all parameters have to use the container class Param. In order to set appropriate values to these parameters, the function *Set_Default_Values()* is called from within the constructor. The function is defined in the protected. The first value on the function gives the default value for the parameter. The second parameter describes the parameter by a string which is used for the output and for detection of the parameter in a configuration structure like XML. The string is required to be set to the name of the variable. The last parameter is a list of object, which is inherently available from the parent class Base_Parameter. Registering each parameter in this list enables some comfort functionalities like printing all parameters by using the streaming operator (<<) on a module object or automatic configuration of the module. The parameter creation, definition of default values and registration can be seen in figure 3.10.

```
public:
      ...
      Param<unsigned int> bw_chv;
      Param<unsigned int> bw_fract;
      Param<unsigned int> num_iter;
      enum DEC_TYPE_ENUM {MIN_SUM, BP};
      Param<DEC_TYPE_ENUM> dec_algo;
       ...
protected:
      virtual void Set_Default_Values()
      {
            bw_chv.Init(6, "bw_chv", param_list_);
            bw_fract.Init(2, "bw_fract", param_list_);
            num_iter.Init(10, "num_iter", param_list_);
            dec_algo.Init(MIN_SUM, "Decoding algorithm", param_list_);
      }
```

Figure 3.10: Excerpt of how to create and define default values to parameters

3.2.2 Choosing or Writing an Interface Class

The framework provides a lot of interface classes for different types of modules. If the new module matches one of these types, it is not needed to write a new interface class. If this is not the case, it is necessary to write a new interface class, but it consists of only a few lines of code. This class defines the input and output ports with the two containers Data_In and Data_Out – figure 3.11 shows the new interface class for a LDPC decoder. Similar to the parameter class, this container provides the "*()*" operator to access the data. The input_data_list and the output_data_list hold all data I/O objects. These lists are necessary for the automatic connection feature of the simulation chain. Naming of the ports has to obey few rules regarding certain prefixes and suffixes that will help the identification of it – for instance, input_prob is the input port that holds probability values or output_bits is the output port that contains information regarding singles bits.

```
#ifndef DEC_LDPC_BINARY_HW_IFACE_H_
#define DEC_LDPC_BINARY_HW_IFACE_H_

class Decoder_LDPC_Binary_HW_Interface : public Base_Interface
{
public:
	Decoder_LDPC_Binary_HW_Interface()
	{
		input_llr.Register("input_llr", input_data_list_);
		output_bits.Register("output_bits", output_data_list_);
	}
	virtual ~Decoder_LDPC_Binary_HW_Interface() {}
	Data_In<int> input_bits_llr;
	Data_Out<unsigned int> output_bits;
};
#endif // DEC_LDPC_BINARY_HW_IFACE_H_
```

Figure 3.11: Code of a new empty interface class for a LDPC decoder

3.2.3 Writing the Module Class Itself

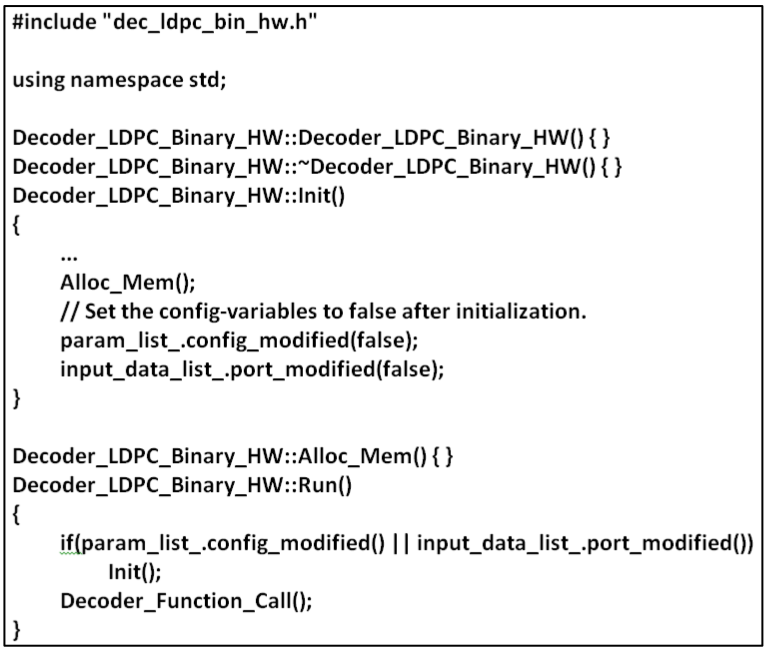
After it was created the parameter class and made sure that we are able to parse the XML file, it is time to write the code for the module. For that it is necessary two files: the header file containing the class definition and the cpp file containing the functionality of the class.

Each module is derived from at least two classes: the interface and the parameter class. The *init()* function sets derived parameters, resizes the outputs ports calling *Aloc_mem()*, and performs other tasks that are required by the module before it is save to invoke the *Run()* function. Now class variables and functions can be defined in the private part of and the according content of the functions can be included in the body of the class. Figure 3.12 shows the header file for a LDPC decoder as an example.

```
#ifndef DEC_LDPC_BINARY_HW_H_
#define DEC_LDPC_BINARY_HW_H_

#include "dec_ldpc_bin_hw_iface.h"
#include "dec_ldpc_bin_hw_param.h"

class Decoder_LDPC_Binary_HW : public Decoder_LDPC_Binary_HW_Interface,
                               public Decoder_LDPC_Binary_HW_Parameter
{
public:
	Decoder_LDPC_Binary_HW()
	virtual ~Decoder_LDPC_Binary_HW();
	int Run();

protected:
	void Init();
	void Alloc_Mem();

private:
  ...
};
#endif
```

Figure 3.12: Code for a new header file for a LDPC decoder

The functions that are part of each module are implemented in the cpp file. A minimum template of a module body is presented in figure 3.13. Important to remind that the if statement in the Run function is mandatory (it allows for an automatic run of the Init function if a parameter or port has changed).



```
#include "dec_ldpc_bin_hw.h"

using namespace std;

Decoder_LDPC_Binary_HW::Decoder_LDPC_Binary_HW() { }
Decoder_LDPC_Binary_HW::~Decoder_LDPC_Binary_HW() { }
Decoder_LDPC_Binary_HW::Init()
{
      ...
      Alloc_Mem();
      // Set the config-variables to false after initialization.
      param_list_.config_modified(false);
      input_data_list_.port_modified(false);
}

Decoder_LDPC_Binary_HW::Alloc_Mem() { }
Decoder_LDPC_Binary_HW::Run()
{
      if(param_list_.config_modified() || input_data_list_.port_modified())
            Init();
      Decoder_Function_Call();
}
```

Figure 3.13: Code for a new cpp file for a LDPC decoder

3.3 General Considerations

Simulations had become an important ease towards the development of any new study (regardless if it is about software or hardware). As first step of this work, it was necessary to understand and to study the functionality of the software CSE and its whole architecture and the group of dynamic and complex tools in order to be able to develop new modules and applications.

All the information linked to the implementation of the new modules related to this work will be presented on chapter 4 – it is important to mention and explain all the mathematical models used on each module: how they were developed, how the class diagrams are, how the data structure and other details are. The main objective with this is to provide the whole information background for a better comprehension of the behavior of each module. The development of the VHDL will also be discussed and explained. It is important and interesting to realize the several differences between the developments of the same module using first C++ and then VHDL.

4 IMPLEMENTATION

The development of new modules is a built-in feature of the CSE. It is a great ease towards the development of new simulations, throughout this work the implementation of a Fine Carrier Synchronization Unit is proposed– which will be completely developed during this work following the method developed on [WASENMÜLLER, 2009]. With this new module it will be possible to decrease the negative influences from noise by calculating the offset parameters – frequency and phase – and correcting. The Fine Carrier Synchronization is a technique that impacts the communication positively by providing ways to do it more accurately, automatically and aiming minimization of errors.

There is also the necessity to develop the Add Offset module and to do some extensions to the Statistics module. The Add Offset module comes with the function to stimulate noises on the system – it adds noise to the bit stream according to the input provided on the XML configuration file. The purpose of the Add Offset module is to check if the Fine Carrier Synchronization module is working properly. The Statistics module had to be extended since it must keep up calculating new probabilistic values regarding the new environment that it would work on - including the Fine Carrier Synchronization and the Add Offset.

The implementation of the Fine Carrier Synchronization Unit in VHDL is also proposed and comes with the idea to provide an embracing approach to the same idea – it is possible to have a good and interesting comparison between two implementations (when designated to different platforms – C++ and VHDL).

It is important to remind that a real communication system will have differences related to the localization and existence of the modules if compared to the software. All the figures developed throughout this work are based on the software disposition of the modules. Therefore, the idea at the moment is to make sure that it is understood the location of it in a real life system. On real life it will exist, for example, a pre amplification module that will be responsible for the empowerment of the signal and the decreasing of the jitter, besides that, the synchronization module will work in parallel to all modules on the receiver part – providing information during all moments to the decoder and the demapper, for example, and not exactly following the linear disposition [ROCHOL, 2011]. This figure 4.1 comes with the purpose to take this idea into consideration – real location of the synchronization module.

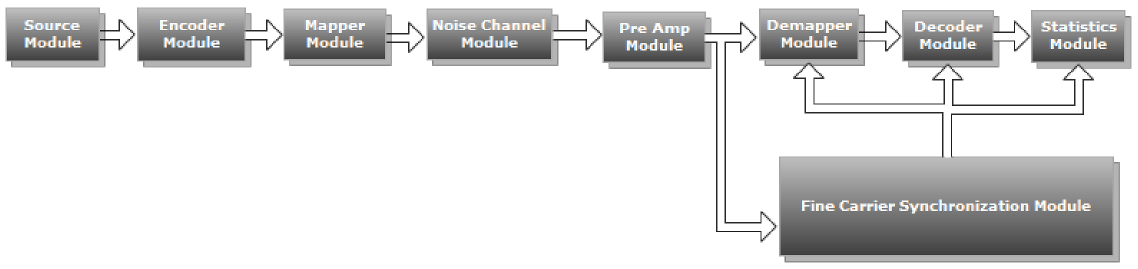


Figure 4.1: Real disposition of the Synchronization module

Due to the software purposes, it will be place in a linear evolution of the information. Therefore, it is located right after the noise channel module. The Fine Carrier Synchronization reacts to every bit stream used on the simulation environment by being located sequentially on the simulation chain path between the Noise Channel Module and the Demapper Module – figure 4.2 shows the location of the new module and certifies that every upcoming bit stream will go through it. Note that this position is valid for test only purposes, in reality this unit is going to be place in reality inside the decoder iteration loop – which will provide the soft information necessary for its functionality.

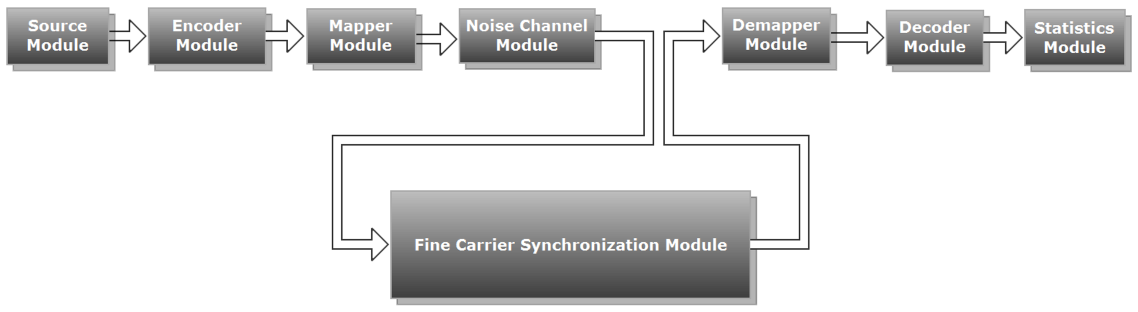


Figure 4.2: Fine Carrier Synchronization Unit Module position regarding the original simulation chain

A proper implementation of the synchronization module can be simplified if the idea of how it is done is divided again into three smaller modules. As shown in figure 4.3, it is possible to recognize subtasks to achieve the objective of this module. By having different and identified subtasks, it is obvious to create sub modules – the subtasks can be easily identified, it makes the design of logic simpler and more accurate. This way, the implemented software will respect some of its principles – reusability, flexibility and object orientation.

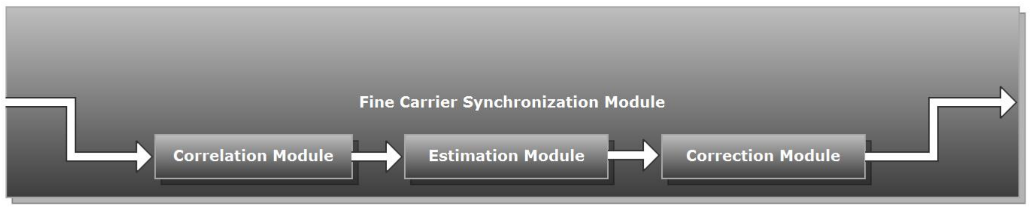


Figure 4.3: Fine Carrier Synchronization Unit Module modularization

The Correlation Module will be responsible for the calculation of the correlation – a statistical measurement of the relationship between the two bit streams – reference and received symbols. The reference symbols are exactly the same bit stream provided by the source module. The received symbols are the bit stream that are being transmitted and had been through the others modules – the encoder, the mapper and the noise channel. The variable that allows the execution of the correlation calculation are the bit streams – the original and the received. Before the Noise Channel module they are perfectly correlated, since they are exactly the same. After the addition of the noise, they are correlated – not perfectly anymore – once the presence of certain characteristics on the original bit stream will react in order to leave traces on the received bit stream. As the Correlation Module produces its output, the correlation value, the Estimation module can do its part – estimate the frequency and phase offset. This is possible based on the average phase of the first and the rear part of the correlation value – when calculating the correlation value it will be divided into two parts in order to provide the values needed for the estimation of the wanted parameters.

Once the value of frequency and phase offset are estimated and available for the next module, the moment when the last task of the Fine Carrier Synchronization Module has come. The Correction module will be responsible for the correction of the noisy signal received in order to decrease the error rates.

For implementation purposes, adaptation and tests of the Fine Carrier Synchronization Module, it is needed a way to provide frequency and phase offset inputs to CSE. This offset is going to be introduced as the last step before the Noise Channel Module. In the end of the simulation it will be possible to compare the values on the XML configuration file and the estimated values of these parameters. The Add Offset module will model the frequency and phase offset at the transmitter side – which means that it will add an error to the bit stream following mathematical background that will be studied also on this work.

During the development of this work, the focus is the implementation of the new modules added on figure 4.4 – Add Offset and Fine Carrier Synchronization. The models used to define parameters on both modules are available and originally on [WASENMÜLLER, 2009]. It is also important to keep in mind that several changes on the Statistics module will also be done in order to embrace the new parameters and functionalities.

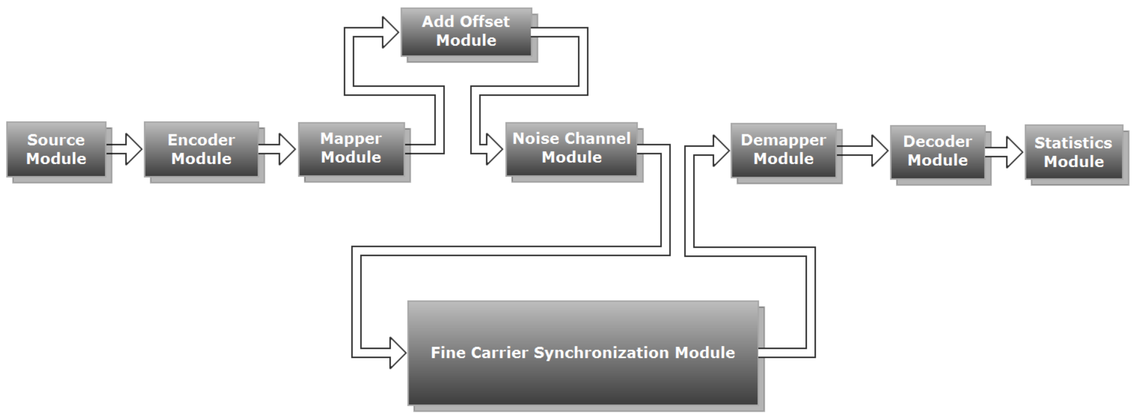


Figure 4.4: Add Offset and Fine Synchronization Module positions regarding the basic simulation chain

This chapter aims to detail the implementation of the new modules for the CSE as well as the corresponding VHDL. The development and implementation comprehend the integration of several basic components and also the extension for some already existing modules.

4.1 Software

The implementation of such functionalities will follow the same language chosen by the original CSE developers – C++, which is a general-purpose programming language with a bias towards system programming that supports efficient low-level computation, data abstraction, object-oriented programming, and generic programming. It provides powerful and flexible mechanisms for abstraction; that is, language constructs that allow the programmer to introduce and use new types of objects that match the concepts of an application. Thus, C++ supports styles of programming that rely on fairly direct manipulation of hardware resources to deliver a high degree of efficiency plus higher-level styles of programming that rely on user-defined types to provide a model of computation that is closer to human's view of the task being performed by a computer. These higher level styles of programming are often called data abstraction, object-oriented programming and generic programming.

4.1.1 Fine Carrier Synchronization Module

The Fine Carrier Synchronization module requires a deep understanding of how it is possible to estimate the parameters and how, once they are known, to perform the elimination of the negative effects that were caused. To make the comprehension easy, the Fine Carrier Synchronization Module will be divided into three other sub-modules; each one of them will be responsible for one task, which will be explained better in the following sections.

By applying modularization it is possible to easily perform changes without requiring modifications in different layers. This simplifies validation, debugging and any other adjustment; it is also highly efficient because of the direct intercommunication between the components. Another benefit of modularization is related to security; it is possible to hide and protect the information that is inside one module since it works as a black box. There will be an interface available for the other modules to communicate with it, not allowing knowing how the functionalities are really implemented.

4.1.1.1 Correlation Module

The correlation module is responsible for the first step done in the whole process performed by the Fine Carrier Synchronization Module. The correlation is one of the most common and most useful statistics. A correlation is a single number that describes the degree of relationship between two variables. The main interest in obtaining the correlation value is to provide the base requirements and parameters to the Estimation module.

$$\widetilde{\phi_k} := \sum_{l=0+k\cdot\frac{L}{2}}^{\frac{L}{2}-1+k\cdot\frac{L}{2}} r(l)\cdot\ s_e^*(l) \qquad k = 0,1 \qquad 4.1$$

On equation 4.1 L represents the length of the burst, r represents the received sample sequence (provided by the channel module) and s_e is the transmitted symbol sequence (provided by the source module) – both sequences are complex values that represent modulated symbols. It is necessary to keep in mind that in reality the s_e is the estimation of the transmitted symbol provided by the decoder – for test purposes it uses the transmitted symbol sequence provided by the source module.

When the correlation value is calculated it is divided into two parts: the front part (with $k = 0$) and the rear part (with $k = 1$), resulting in the values of $\tilde{\phi}_0$ *and* $\tilde{\phi}_1$.

It is interesting to keep in mind the division of the correlation value in two (front part and rear part). This can be seen clearly in figure 4.5, which shows the implementation of the correlation values, by the comparison of the iteration index value (i) with the length of the burst $(source\_symb().length())$.

```
for(unsigned int i = 0; i < source_symb().length(); i++){
	k = k + (complex<int64_t>)(received_symb()[i] * conj(source_symb()[i]));
	if (i==(source_symb().length()-1)/2)
		k_0 = k;
}
```

Figure 4.5: Implementation of the calculation of the correlation values

The class diagram related to the implementation of the Correlation module is shown in figure 4.6.

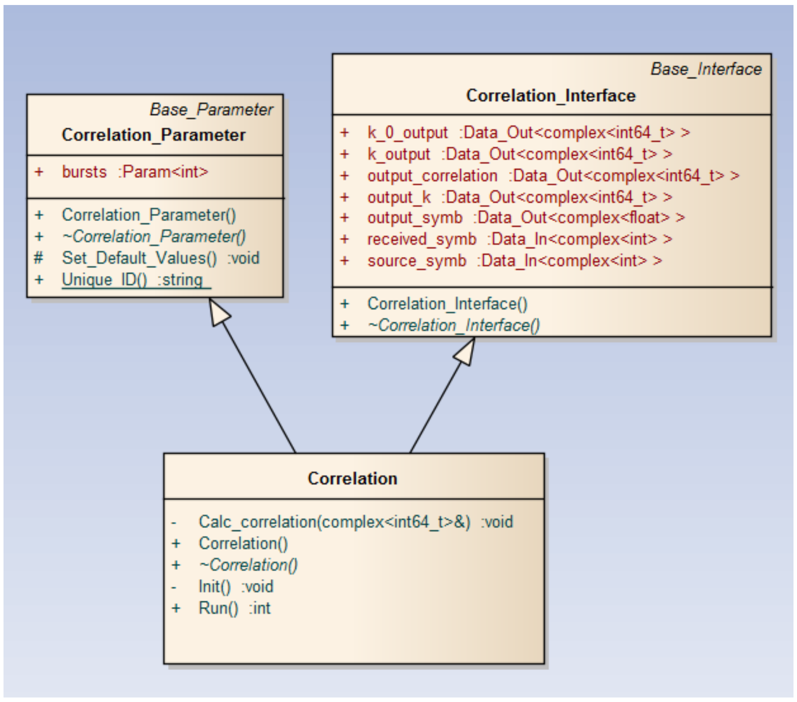

Base_Parameter
Correlation_Parameter
+ bursts :Param<int>
+ Correlation_Parameter()
+ ~Correlation_Parameter()
# Set_Default_Values() :void
+ Unique_ID() :string
Base_Interface
Correlation_Interface
+ k_0_output :Data_Out<complex<int64_t> >
+ k_output :Data_Out<complex<int64_t> >
+ output_correlation :Data_Out<complex<int64_t> >
+ output_k :Data_Out<complex<int64_t> >
+ output_symb :Data_Out<complex<float> >
+ received_symb :Data_In<complex<int> >
+ source_symb :Data_In<complex<int> >
+ Correlation_Interface()
+ ~Correlation_Interface()
Correlation
- Calc_correlation(complex<int64_t>&) :void
+ Correlation()
+ ~Correlation()
- Init() :void
+ Run() :int

Figure 4.6: Correlation module class diagram

4.1.1.2 Estimation Module

With the $\tilde{\phi}_{0,1}$values available it is possible to proceed with the estimation of the frequency and phase offset values. The estimation of the frequency offset can be calculated with the equation 4.2 and the phase offset estimation with the equation 4.3.

$$\tilde{f}_0 = \frac{\arg(\tilde{\phi}_1 \cdot \tilde{\phi}_0)}{\pi \cdot L} \qquad (4.2)$$

$$\tilde{\phi} = \arg\left(\widetilde{\phi_0} + \widetilde{\phi_1}\right) - L \cdot \tilde{f_0} \cdot \pi \tag{4.3}$$

Figure 4.7 shows the implementation of the estimation module based on the values of the correlation module (output from equation 4.1). In equation 4.2 the function “arg ()” returns the phase angle of the multiplication from the two correlation values.

```
//Correlation total value
k_total =k;
k_1 = k_total-k_0;
// Argument of mult and sum in scaled radians
mult=k_1;
mult=mult*conj(k_0);

// transforms scaled radians (9 bits) to radians
arg_mult=rad_to_scaled_radians(mult,9);
// transforms scaled radians (9 bits) to radians
arg_sum=rad_to_scaled_radians(k_total,9);

// Calculation of the estimated frequency and phase offset accordingly to the paper, chapter 3.
freq_est=scaled_radians_to_rad(freq_estimation(arg_mult,L),9);
phase_est=scaled_radians_to_rad(phase_estimation(arg_sum,arg_mult),9);

float freq_estimation(float arg_mul,int L)
{
      return arg_mul*2/L;
}
float phase_estimation(float arg_sum, float arg_mult)
{
      return arg_sum-arg_mult;
}
```

Figure 4.7: Implementation of the estimation frequency and phase offset

The class diagram related to the implementation of the Estimation module is shown in figure 4.8.

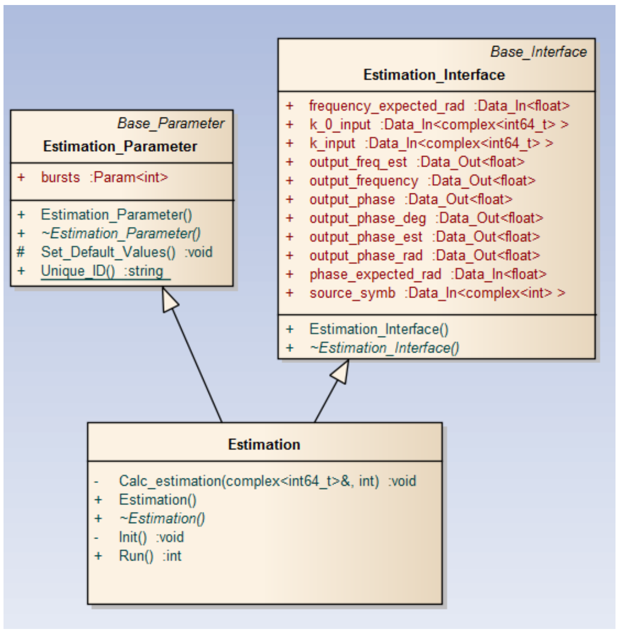

Base_Parameter
Estimation_Parameter
+ bursts :Param<int>
+ Estimation_Parameter()
+ ~Estimation_Parameter()
# Set_Default_Values() :void
+ Unique_ID() :string
Base_Interface
Estimation_Interface
+ frequency_expected_rad :Data_In<float>
+ k_0_input :Data_In<complex<int64_t> >
+ k_input :Data_In<complex<int64_t> >
+ output_freq_est :Data_Out<float>
+ output_frequency :Data_Out<float>
+ output_phase :Data_Out<float>
+ output_phase_deg :Data_Out<float>
+ output_phase_est :Data_Out<float>
+ output_phase_rad :Data_Out<float>
+ phase_expected_rad :Data_In<float>
+ source_symb :Data_In<complex<int> >
+ Estimation_Interface()
+ ~Estimation_Interface()
Estimation
- Calc_estimation(complex<int64_t>&, int) :void
+ Estimation()
+ ~Estimation()
- Init() :void
+ Run() :int

Figure 4.8: Estimation module class diagram

4.1.1.3 Correction Module

Once the value of the frequency and phase offset are estimated and available, when the burst has come to its end, it is possible to perform the correction itself.

$$r_{corrected}(l) = r(l) \cdot e^{(phase_{estimation}+frequency_{estimation}*l\,)*(-j)} \qquad (4.4)$$

The correction will be done according to the equation 4.4, where $r_{corrected}$ represents the corrected symbols sequence, r represents the original sample sequence (the sequence that has been affected by the noise and contains errors), $phase_{estimation}$ and $frequency_{estimation}$ are the respective estimated offsets – coming from estimation module and j represents the imaginary term of a complex number.

The equation 4.4 is responsible for the elimination of the negative effects of phase and frequency offset – it is basically the opposite function of what happens in real world when these parameters are added to the communication system.

The class diagram related to the implementation of the Estimation module is shown in figure 4.9.

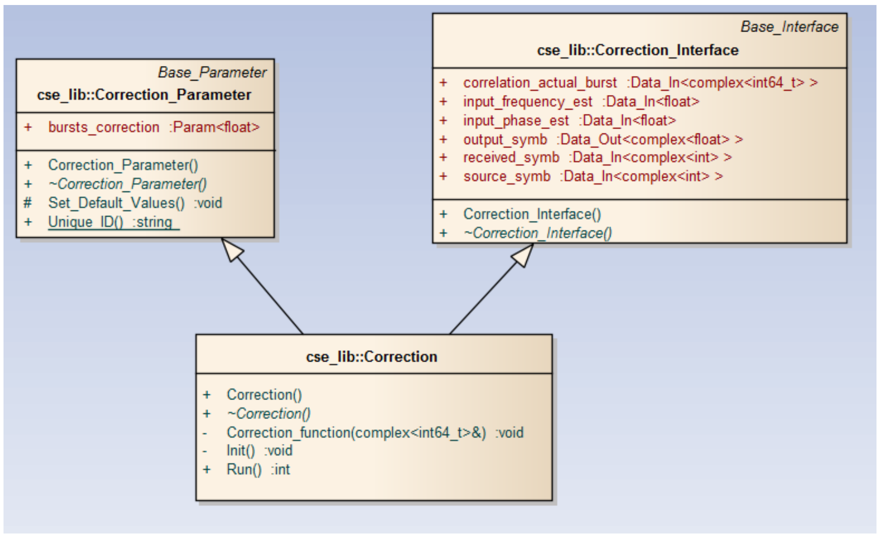

Base_Parameter
cse_lib::Correction_Parameter
+ bursts_correction :Param<float>
+ Correction_Parameter()
+ ~Correction_Parameter()
# Set_Default_Values() :void
+ Unique_ID() :string
Base_Interface
cse_lib::Correction_Interface
+ correlation_actual_burst :Data_In<complex<int64_t> >
+ input_frequency_est :Data_In<float>
+ input_phase_est :Data_In<float>
+ output_symb :Data_Out<complex<float> >
+ received_symb :Data_In<complex<int> >
+ source_symb :Data_In<complex<int> >
+ Correction_Interface()
+ ~Correction_Interface()
cse_lib::Correction
+ Correction()
+ ~Correction()
- Correction_function(complex<int64_t>&) :void
- Init() :void
+ Run() :int

Figure 4.9: Correction module class diagram

4.1.1.4 Implementation Details

During the implementation of the Estimation module, the “Wrap Around Problem” was faced. This problem happens due to the software interpretation of the borders when calculating values of frequency and phase offset to perform the correction and to put into the accumulator, which will have central importance in the Statistics Module.

When calculating probabilistic values related to the estimation of the phase and frequency offset, like mean and standard deviation, it is necessary to take a careful approach to symbols like 0.1radians and 2π+0.1radians. They are considered completely different but they are actually the same. For example, in figure 4.10 nearly the same symbol can have different wrong interpretations since the fluctuation of the symbol can vary between the upper and lower part of real axis (Re). The wrong interpretation of it happens since the software does not take it as a circular unit – therefore the value 2π-0.1 will be “far” from the 0.1, resulting in several errors when calculating the accumulate value of offsets and also regarding the statistics values; the mean and standard deviation of the estimation will have drastically wrong results.

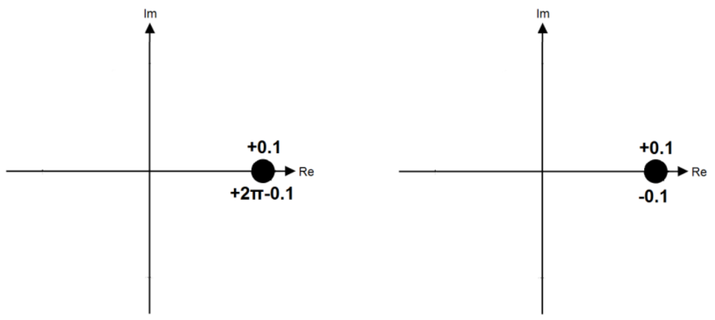


Figure 4.10: Example of the “Wrap Around Problem” with the wrong interpretation (left) and the right interpretation (right)

Therefore, a normalization of the results that are out of bounds is mandatory after every estimation. The range of accepted estimation phase offset goes from [-π,+π] and the range of accepted estimation frequency offset from [-0.5,+0.5] – anything out of this range must be normalized. The normalization is done by subtracting the expected phase or frequency offset from the estimation value. To this new value, 2π (on the phase estimation) or 1 (on the frequency estimation) is added, when it is in the negative area considered out of bounds (identified by “NEG” in figure 4.11), or subtracted, when it is in the positive area considered out of bounds (identified by “POS” in figure 4.11) – this can be seen easily with the part of the code responsible for this normalization shown in figure 4.12. After doing this, the value is added to the expected phase or frequency (which is available from the XML configuration file) – this final value contains the normalized estimation of frequency or phase offset.

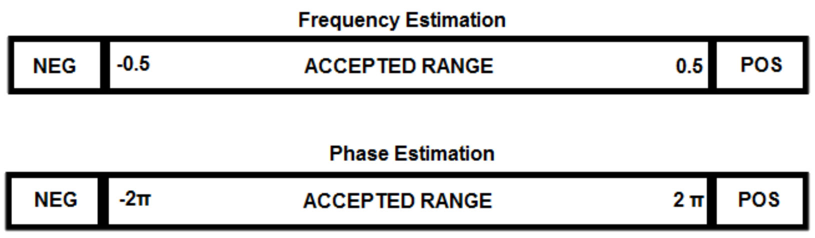


Figure 4.11: Illustration of accepted range and “out of bounds” areas regarding frequency and phase estimation

```
// Normalization of results that are out of
// bounds (range of the estimated phase [-pi,+pi]
diff = phase_est-phase_expected;
if (diff<-pi)
      diff = diff+2*pi;
if (diff>pi)
      diff = diff-2*pi;
diff += phase_expected;
phase_est=diff;

// Normalization of results that are out of
// bounds (range of the estimated phase [-0.5,+0.5]
diff= freq_est-frequency_expected;
if (diff<-0.5)
      diff = diff+1;
if (diff>0.5)
      diff = diff-1;
diff += frequency_expected;
freq_est=diff;
```

Figure 4.12: Excerpt of the code used for the normalization of estimated values

4.1.2 Add Offset Module

The objective is to introduce, in an artificial way, errors on the communication system to evaluate and measure the quality and the behavior of the Fine Carrier Synchronization module.

The mathematical model that introduces phase and frequency offset into the burst is strategically located before the Channel Module – it is the last step before the burst goes into "real world".

$$r_{with\ offset}\ (l) = \ r(l) * e^{(phase_{offset}+frequency_{offset}*l*2*\pi\ )*(j)} \qquad (4.5)$$

The offset will be added according to the equation 4.5, where $r_{with\ offset}(l)$ represents the symbols with offset, $r(l)$ represents the original symbol – error free, $phase_{offset}$ and $frequency_{offset}$ are the respective offsets which will be introduced according to the information on the XML configuration file and j represents the imaginary term of a complex number.

The values of phase and frequency offset that can be defined on the XML input file must be within a certain range of acceptable offset values; if the values do not obey certain rules it will be impossible to estimate and correct it properly – making the job of the Fine Carrier Synchronization Unit impossible. The frequency offset, for example, value must respect equation 4.6. The phase offset works in a 2π complex plane which automatically limits the maximal value to the circumference; therefore, adding a phase offset of 3π will be the same as adding π.

$$frequency_{offset} * L * 2\pi \leq \ (\frac{\pi}{2}) \qquad (4.6)$$

The class diagram related to the implementation of the Add Offset module is shown in figure 4.13.

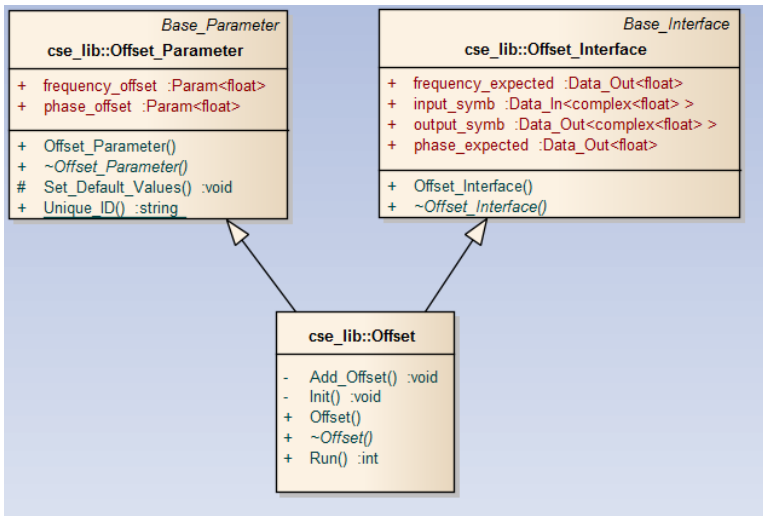

Base_Parameter
cse_lib::Offset_Parameter
+ frequency_offset :Param<float>
+ phase_offset :Param<float>
+ Offset_Parameter()
+ ~Offset_Parameter()
# Set_Default_Values() :void
+ Unique_ID() :string
Base_Interface
cse_lib::Offset_Interface
+ frequency_expected :Data_Out<float>
+ input_symb :Data_In<complex<float> >
+ output_symb :Data_Out<complex<float> >
+ phase_expected :Data_Out<float>
+ Offset_Interface()
+ ~Offset_Interface()
cse_lib::Offset
- Add_Offset() :void
- Init() :void
+ Offset()
+ ~Offset()
+ Run() :int

Figure 4.13: Add Offset module class diagram

4.1.3 Extensions to Statistics Module

This module was the only one already presented on the original CSE that was changed in order to adapt itself to the new functionalities after the Fine Carrier Synchronization Module was included. These changes were necessary once the statistics module should be also responsible for several quality measurements and statistics calculations regarding the new values and parameters included in the new modules – i.e. correlation values and estimation of the frequency and phase offset.

The first step was to create the several new interfaces; the statistics module now needs to receive data values from the Add Offset, Correlation and Estimation modules. Important to remind that on the first version of the Statistics module there were just two modules submitting input information: the Demapper and the Source Bits.

The Statistics Module has as part of its job the creation of the XML output file, which contains the original configuration of the simulation (it is a transcription of the configuration found on the XML input file) and the results of the calculation of the statistics themselves.

On the original XML output file 6 outputs and the configuration used for the simulation were available, specifying which modules were used and how the parameters were defined. The new Statistics Module has 18 new outputs, which will provide a good analysis about data included on the new modules. This means that now the output file contains 24 output values, providing information that can easily be analyzed and used for aiming improvements in the quality of the software developed. Some of them are just conversions between units (radians to degrees and vice versa).

Among the new statistics module calculations, it is important to keep in mind the basic probability equations as the calculation of the frequency and phase estimation and the variance/sigma of the phase and frequency (expected and estimated). Several equations are going to be omitted here since there are just changes on the data measured (for example frequency offset estimation instead of phase offset estimation – as shown in equations 4.7 and 4.8). The comprehension of the whole system and analysis of the quality of the new functionalities implemented are based on the understanding of these steps – the output of the Statistics Module must be analyzed deeply and interpreted.

$$Variance\ of\ Phase\ Estimation = \frac{1}{L}\sum_{i=0}^{L-1}\left(PhaseEstimation_i - PhaseEstimation_{average}\right)^2 \tag{4.7}$$

$$Variance\ of\ Phase\ Estimation\ Only = \frac{1}{L}\sum_{i=0}^{L-1}\left(PhaseOnly_i - PhaseOnly_{average}\right)^2 \tag{4.8}$$

Where

$$PhaseOnly_i = \tan^{-1}\left(\frac{K_{imag\ i}}{K_{real\ i}}\right) \tag{4.9}$$

In equations 4.7, 4.8 and 4.9, L represents the number of bursts, K represents the correlation. The differences between equation 4.7 and 4.8 is that the first takes into account the estimation of the frequency offset when estimating the phase offset, while the second estimate the phase only based on the correlation of the burst.

Important to keep in mind that all the calculations done in this module are only possible if the “Wrap Around Problem” was successfully corrected, otherwise problems with the values of variance and mean will be inevitable.

The class diagram related to the implementation of the Statistics module is shown in figure 4.14.

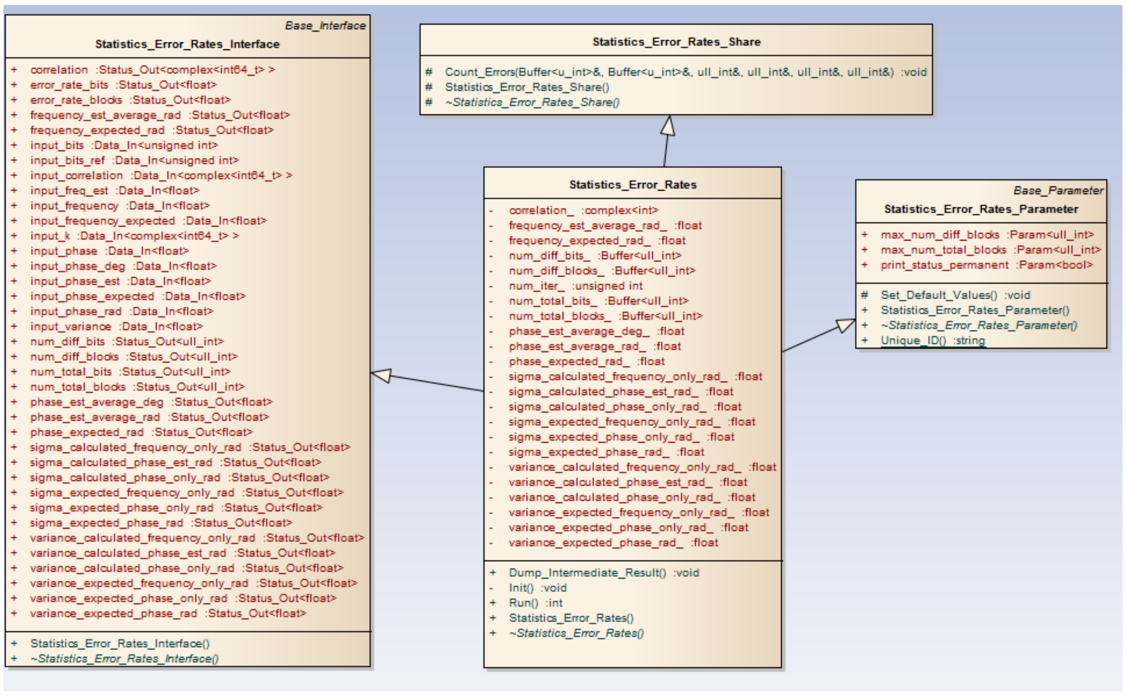

Base_Interface
Statistics_Error_Rates_Interface
+ correlation :Status_Out<complex<int64_t> >
+ error_rate_bits :Status_Out<float>
+ error_rate_blocks :Status_Out<float>
+ frequency_est_average_rad :Status_Out<float>
+ frequency_expected_rad :Status_Out<float>
+ input_bits :Data_In<unsigned int>
+ input_bits_ref :Data_In<unsigned int>
+ input_correlation :Data_In<complex<int64_t> >
+ input_freq_est :Data_In<float>
+ input_frequency :Data_In<float>
+ input_frequency_expected :Data_In<float>
+ input_k :Data_In<complex<int64_t> >
+ input_phase :Data_In<float>
+ input_phase_deg :Data_In<float>
+ input_phase_est :Data_In<float>
+ input_phase_expected :Data_In<float>
+ input_phase_rad :Data_In<float>
+ input_variance :Data_In<float>
+ num_diff_bits :Status_Out<ull_int>
+ num_diff_blocks :Status_Out<ull_int>
+ num_total_bits :Status_Out<ull_int>
+ num_total_blocks :Status_Out<ull_int>
+ phase_est_average_deg :Status_Out<float>
+ phase_est_average_rad :Status_Out<float>
+ phase_expected_rad :Status_Out<float>
+ sigma_calculated_frequency_only_rad :Status_Out<float>
+ sigma_calculated_phase_est_rad :Status_Out<float>
+ sigma_calculated_phase_only_rad :Status_Out<float>
+ sigma_expected_frequency_only_rad :Status_Out<float>
+ sigma_expected_phase_only_rad :Status_Out<float>
+ sigma_expected_phase_rad :Status_Out<float>
+ variance_calculated_frequency_only_rad :Status_Out<float>
+ variance_calculated_phase_est_rad :Status_Out<float>
+ variance_calculated_phase_only_rad :Status_Out<float>
+ variance_expected_frequency_only_rad :Status_Out<float>
+ variance_expected_phase_only_rad :Status_Out<float>
+ variance_expected_phase_rad :Status_Out<float>
+ Statistics_Error_Rates_Interface()
+ ~Statistics_Error_Rates_Interface()
Statistics_Error_Rates_Share
# Count_Errors(Buffer<u_int>&, Buffer<u_int>&, ull_int&, ull_int&, ull_int&, ull_int&) :void
# Statistics_Error_Rates_Share()
# ~Statistics_Error_Rates_Share()
Statistics_Error_Rates
- correlation_ :complex<int>
- frequency_est_average_rad_ :float
- frequency_expected_rad_ :float
- num_diff_bits_ :Buffer<ull_int>
- num_diff_blocks_ :Buffer<ull_int>
- num_iter_ :unsigned int
- num_total_bits_ :Buffer<ull_int>
- num_total_blocks_ :Buffer<ull_int>
- phase_est_average_deg_ :float
- phase_est_average_rad_ :float
- phase_expected_rad_ :float
- sigma_calculated_frequency_only_rad_ :float
- sigma_calculated_phase_est_rad_ :float
- sigma_calculated_phase_only_rad_ :float
- sigma_expected_frequency_only_rad_ :float
- sigma_expected_phase_only_rad_ :float
- sigma_expected_phase_rad_ :float
- variance_calculated_frequency_only_rad_ :float
- variance_calculated_phase_est_rad_ :float
- variance_calculated_phase_only_rad_ :float
- variance_expected_frequency_only_rad_ :float
- variance_expected_phase_only_rad_ :float
- variance_expected_phase_rad_ :float
+ Dump_Intermediate_Result() :void
- Init() :void
+ Run() :int
+ Statistics_Error_Rates()
+ ~Statistics_Error_Rates()
Base_Parameter
Statistics_Error_Rates_Parameter
+ max_num_diff_blocks :Param<ull_int>
+ max_num_total_blocks :Param<ull_int>
+ print_status_permanent :Param<bool>
# Set_Default_Values() :void
+ Statistics_Error_Rates_Parameter()
+ ~Statistics_Error_Rates_Parameter()
+ Unique_ID() :string

Figure 4.14: Correction module class diagram

4.2 Hardware

The implementation of the Fine Carrier Synchronization Unit in VHDL requires a view of the functionalities in a low level of abstraction; it is necessary, for example, to organize signals that will be responsible of the read/write of the ROM memory – when regarding software development, this kind of control is completely unimportant since it is done automatically. On the other hand, some other features, for example, the implementation of the Add Offset Module is completely ignored when regarding VHDL since the “real world” will be in charge of the Add Offset Module’s function – stimulate different – phase or/and frequency – noises on the system’s communication.

The implementation of such functionalities is done using VHDL – a brief discussion about this hardware description language is also introduced in this work to provide a basic knowledge about a subject that is of primary importance regarding the evolution of this work. Fine Carrier Synchronization Unit was not implemented on a FPGA due to resources reasons but once the final developed VHDL is synthesizable – and this is a big concern throughout this work – it is possible to implement the hardware with such behavior.

4.2.1 VHDL Overview

VHDL is a nested acronym that stands for Very High Speed Integrated Circuits (VHSIC) Hardware Description Language. VHDL allows a view of a design at various levels of abstraction – Simulation gives the waveforms of the circuit inputs and outputs and Synthesis gives the possible combination of gates/transistors to achieve the required operation [VHDL, 2012].

In VHDL, any circuit/system is viewed as an entity (or a set of entities). The internal working of an entity is called its architecture. For instance, to design a half adder using VHDL, the entity would be the half adder itself with its input and output ports and the architecture would tell VHDL what happens between the input and the output ports.

VHDL has libraries that allow the reuse important and frequently used pieces of code. All packages contain useful data types and keywords. In VHDL, data can be in the form of a Constant, Variable or a Signal – which are also the keywords for declaring the same. “File” in VHDL is a sequence of values and hence, is also considered a data object.

Alignment operators are used to assign a value to a data object and there are three types: “<=” assigns a value to a signal; “:=” assigns a value to a variable and “=>” used to assign values to individual/other vector elements. Logical operators respect the Boolean logic; to operate with this logic the data must be of type BIT, STD_LOGIC or STD_ULOGIC (or their vector extensions). Arithmetic operators (+, -, *, /, **, MOD, REM and ABS), relational operators (=, /=, >, <, <= and >=) and shift operators (sll, slr, sla, slr, rol and ror) are also present.

Another VHDL feature that can be very useful are the Generics, which allow a design entity to be described so that, for each use of that component, its structure and behavior can be changed by generic values. In general, they are used to construct parameterized hardware components and can be of any type. Generic is a great asset when the design has slight changes at many places, change in the register sizes, input sizes, etc. If the design is very unique then, there is no need to have generic parameters.

Unlike the sequential statements in other programming languages, VHDL code is concurrent code – which is good enough to build combinational circuits. However, to build sequential circuits, we need sequential code. To write concurrent code, use WHEN, GENERATE and BLOCK statements and to write sequential code, use the PROCESS, FUNCTION and PROCEDURE statements.

The processing of a VHDL code occurs in three stages:

- Analysis: compiler checks each design unit for correct syntax and for some static semantic errors; if no errors are found, the compiler translates the unit into an intermediate form and stores it in a designated library.
- Elaboration: binds architectures to entities using configuration data. Many complex designs are coded in a hierarchical manner. Compiler starts with designated top-level component and replaces all instantiates sub-components with their architecture bodies to create a single flattened description.
- Execution: the flattened design is used as input to a simulation or synthesis engine.

Regarding hardware descriptions languages, synthesis is a process where the code is compiled and mapped into an implementation technology such as an field-programmable gate array (FPGA) or an application-specific integrated circuit (ASIC). It is important to keep in mind that not all constructs are suitable for synthesis. For example, constructs related to timing are valid for simulation but not synthesizable.

During the implementation of this work, the aim to develop a synthesizable hardware at the end of this project was taken as a primary goal. This was possible with the framework used throughout this work, which was Xilinx ISE Design Suite 13.2 [XILINX, 2012]. It comes with a series of in-built tools allowing that everything can be done on the same framework (from design entry, through implementation and verification, to device programming from within the unified environment of the ISE Design Suite). As alternative synthesis tools available on the market nowadays can be mentioned Simplify Pro and Leonardo Spectrum.

4.2.2 Fine Carrier Synchronization Unit

The simulator used on the first moment of development was the ModelSim Simulator, which is not included on the Xilinx ISE tools [MODELSIM, 2012]. Due to reasons such as problems with the compatibility with the several versions of the Xilinx ISE Design Suite, the simulator had to be changed. It was then, preceded with the simulator included on the framework, already a built-in feature of the Xilinx ISE Design Suite, called ISim [XILINX ISIM, 2009].

The intention at all moments during VHDL development was to implement a synthesizable VHDL. By using this term, synthesizable, it refers to the capability of the synthesis tool to implement the given program in hardware. With other words, it is intended to at the end of the development have something that "can be transformed into physical part" by a synthesis tool chosen. Whether a particular VHDL statement is synthesizable or not depends on the technology planned to realize the final "physical part". This work aims the technology family Virtex 6, device XC6VLX75T, package FF484 and speed -1. The synthesis tool used is the XST and the simulator used is ISim (synthesis and simulator are built-in features of the framework).

It is important to understand that now a bottom-up approach will be used to comprehend exactly all the modules functionalities. The same idea used before during the software development will be taken now: modularization of the whole unit accordingly to its tasks; sub-tasks inside sub-modules makes the work easier and less impacting.

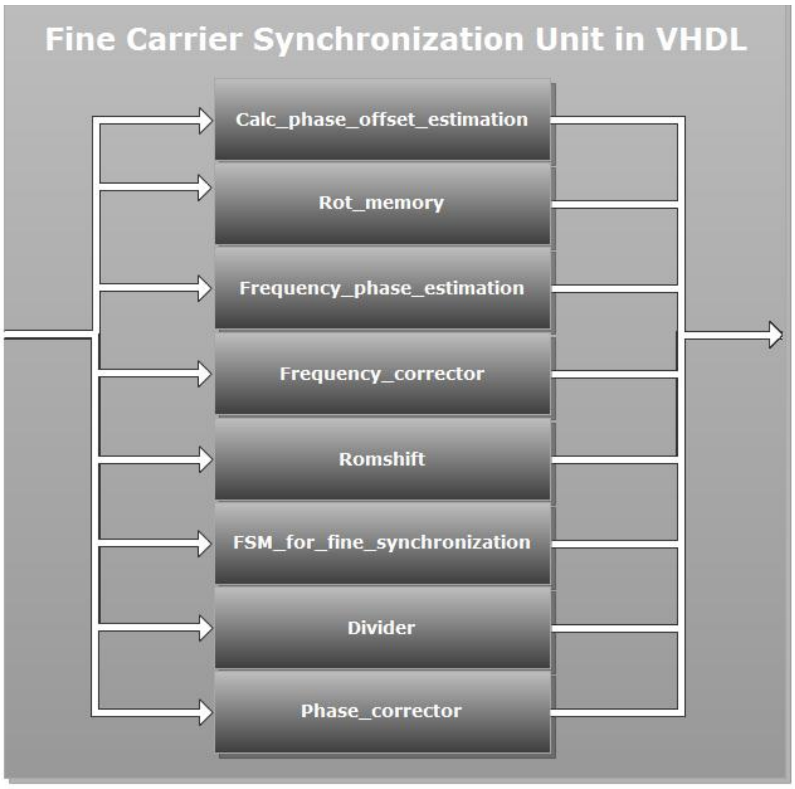

Fine Carrier Synchronization Unit in VHDL
Calc_phase_offset_estimation
Rot_memory
Frequency_phase_estimation
Frequency_corrector
Romshift
FSM_for_fine_synchronization
Divider
Phase_corrector

Figure 4.15: Fine Carrier Synchronization Unit in VHDL modularization

Fine Carrier Synchronization implementation in VHDL was organized keeping in mind the same approach adopted when implementing the software, modularization and division of tasks – this is shown in figure 4.15. Several modules were necessary – each one has a specific and well defined function. This set of modules need to communicate with each other at all moments during the execution. Important to mention that there are three kind of modules used during this work: completely implemented during this work, implemented by co-researchers and generated with a Xilinx ISE Design Suite in-built tool called Core Generator.

Calc_phase_offset_estimation module is responsible to perform the calculation of the correlation. The basic of the correlation calculation is a multiplication between the two sets of complex values, which is not a trivial task in VHDL. Figure 4.16 shows the main function of this module – two accumulators were created to hold the results of the complex multiplication and a selector to define whether it belongs to the front or the rear part of the burst. Calc_phase_offset_estimation was completely developed during this work.

```
if (index = 0 or index <= (index_half -1)) then
      ac_0 <= (signed(received_symb_real) * signed(reference_symb_real)) + ac_0;
      bd_0 <= (signed(received_symb_imag) * signed(reference_symb_imag)) + bd_0;
      ad_0 <=   (signed(received_symb_real) * signed(reference_symb_imag)) + ad_0;
      bc_0 <= (signed(received_symb_imag) * signed(reference_symb_real)) + bc_0;
else
      if (index_half <= (index) and (index) <= (signed(index_in) -1)) then
            ac_1 <= (signed(received_symb_real) * signed(reference_symb_real)) + ac_1;
            bd_1 <= (signed(received_symb_imag) * signed(reference_symb_imag)) + bd_1;
            ad_1 <=   (signed(received_symb_real) * signed(reference_symb_imag)) + ad_1;
            bc_1 <= (signed(received_symb_imag) * signed(reference_symb_real)) + bc_1;
      end if; --((index_half <= index) and (index <= signed(index_in) -1)) then
end if; --(index = 0 or index=1  or index <= (index_half -1)) then
```

Figure 4.16: Excerpt of the correlation calculation in VHDL

FSM_for_fine_synchronization module is a finite state machine which takes the responsibility over signal that controls the “correction module”. It will wait until both frequency and phase offset estimation values are ready and available and then send the “start” signal for the correction. It is basically the trigger for the correction itself of the symbols. FSM_for_fine_synchronization was completely developed during this work.

Frequency_corrector module adjusts the correction value after every iteration during the whole burst – the frequency offset must be added after every iteration (this will correspond to the multiplication by the number of symbols, represented in equation 4.4 by l). During the development of this module a problem was faced - the original phase and frequency offset were made with a different bit width since the frequency offset has a fractional part – it was necessary, therefore, to create a fractional part for the phase estimation (always defined with zero as its value) to add values with the same number of bits. Frequency_corrector was completely developed during this work.

Frequency_phase_estimation module is in charge of the validation of the phase and frequency estimation values. Since the divider module takes a bit longer than others to produce its output it is necessary that others module wait for it, once the result of the division entails directly on further computations – as it can be seen on figure 4.17. Frequency_phase_estimation it was also completely developed during this work.

```
if validator_sig = '1' then
            if((ready_for_data_sum ='1')and (ready_for_data_mult='1'))  then
                  if division_ready = '1' then
                  last<=true;
                  validator_sig <= '0';
            end if;
            frequency_offset_estimation_valid<='1';
            division_ready<='1';
            phase_offset_estimation_valid<='1';
            frequency_offset_real<=scaled_radiance_mult &'0'; -- scaled_radiance_mult * 2
            phase_offset_real <= std_logic_vector(signed(scaled_radiance_sum) - signed(scaled_radiance_mult));
      end if; -- ready_for_data_sum ='1')and (ready_for_data_mult='1'))
end if; --validator_sig = '1'
```

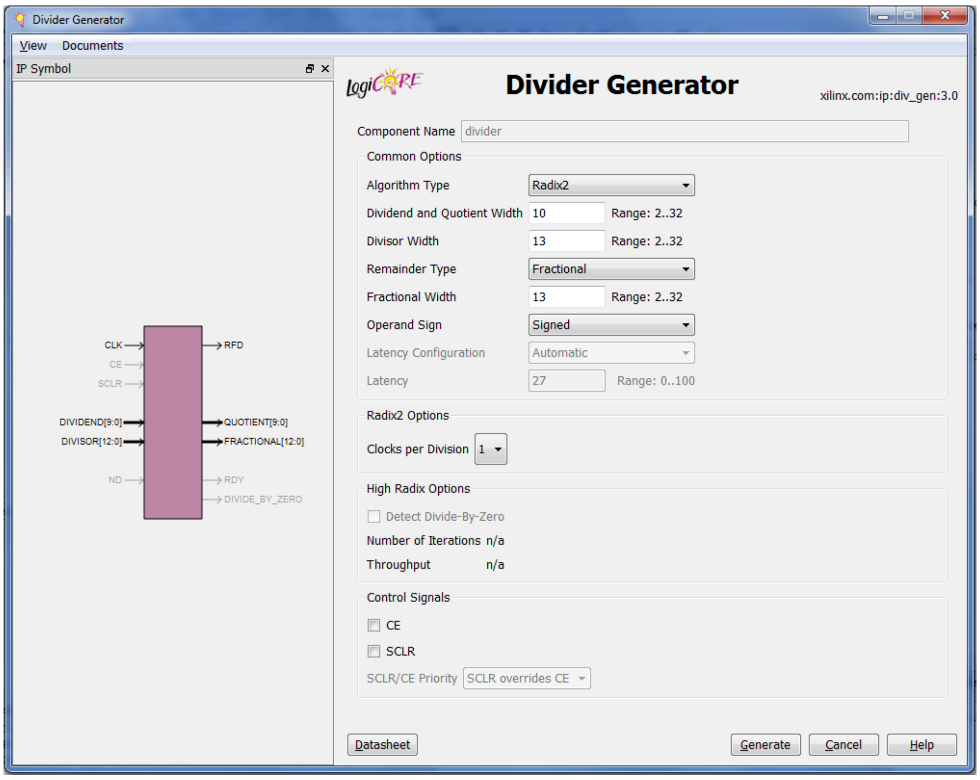
Figure 4.17: Excerpt of the calculation

Phase_corrector module is responsible for the correction itself, this module will receive the phase and frequency estimated offset and do the multiplication. This is improved by using a look up table containing the values for sines and cosines created with Core Generator from Xilinx – since the calculation of sin and cosine in VHDL is quite complicated to be done, it was chosen to use a look up table to speed up the whole process. Phase_corrector module was developed by Robert Drachenberg and Uwe Wasenmüller.

The Romshift module is performing the function that provide as output an approximated scaled radiance in 9 bit length for complex input of generic length. This module has two sub modules inside it: the first is a read-only-memory (ROM) contain the instantiation of an array for 4096 angles for 12 bit complex numbers (6 bit real part and 6 bit imaginary part); the second is a sub module responsible for to round complex signals with the user defined length to a wanted length (Smartscale_generic_shift). All the parts of this module were developed by Paul Salzmann, Harald Schenk and Uwe Wasenmüller.

Rot_memory module is a memory for the storage of symbols while the calculations are being done. The rot_memory is also responsible for re-sending the burst after the whole calculation is done; during the calculation of the estimation it just stand by and when everything is ready to start the correction this module re-sends the received symbols at the same order that they were received. This module was created using the Xilinx tool Core Generator [XILINX CORE, 2012].

Divider module implements the division on VHDL – which is necessary in order to estimate the frequency offset that was introduced to the symbols. Division in VHDL is a topic with a very complicated approach; therefore, to assure reliability on this task and not to spend time reworking on something, this module was created also using the Core Generator. The interface this tool used for creating and editing the details of the module can be seen on figure 4.18.



Divider Generator xilinx.com:ip:div_gen:3.0

Component Name: divider

Common Options
Algorithm Type: Radix2
Dividend and Quotient Width: 10 Range: 2..32
Divisor Width: 13 Range: 2..32
Remainder Type: Fractional
Fractional Width: 13 Range: 2..32
Operand Sign: Signed
Latency Configuration: Automatic
Latency: 27 Range: 0..100

Radix2 Options
Clocks per Division: 1

High Radix Options
☐ Detect Divide-By-Zero
Number of Iterations: n/a
Throughput: n/a

Control Signals
☐ CE
☐ SCLR
SCLR/CE Priority: SCLR overrides CE

Figure 4.18: GUI interface of the Xilinx tool Core Generator when creating and editing the Divider module

4.2.3 Implementation Details

A Xilinx tool known as Core Generator (already included on the design suite ISE 13.2) was used during the development of the modules divider and rot_symbols. This tool was used aiming specially for the advantage given by the acceleration of the design time on the access provided to highly parameterized Intellectual Properties (IP), which means in other words, by using modules that are built and can be configured accordingly to the needs of the project with this tool. Core Generator provides a catalog of architecture specific, domain-specific and market specific IP. These user-customizable IP functions range in complexity from commonly used functions, such as memories to system-level building blocks, such as filters and transforms. Using these IP blocks can save days to months of design time and the range of applications is really impressive, as shown in figure 4.19. The IP blocks are already developed and they are completely configurable – which will fit probably to 100% of the user cases.

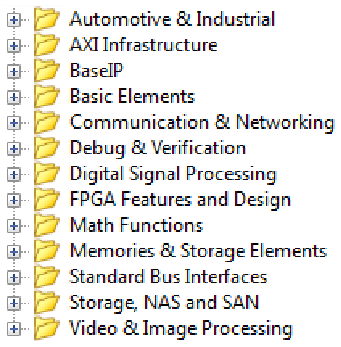
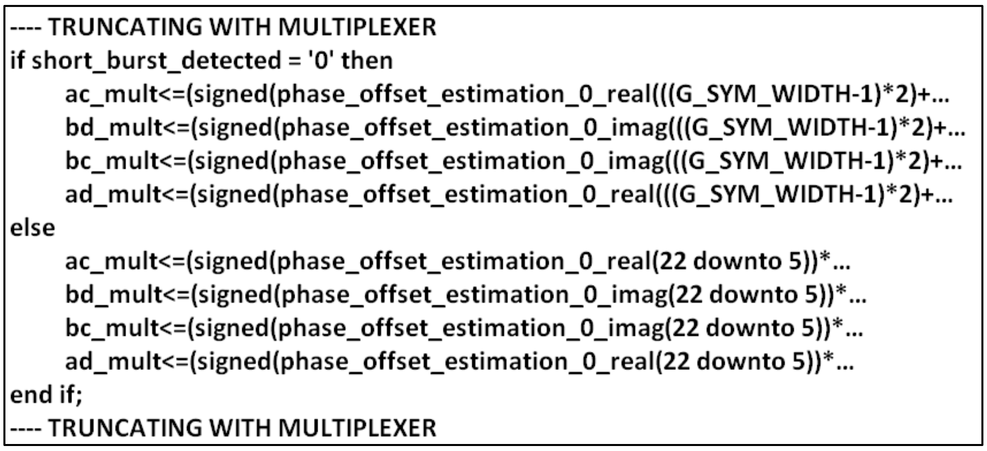


Figure 4.19: Xilinx Core Generator IP Catalog

During the implementation of the work, an important feature of VHDL - Generics - was used. Generics allow a design entity to be described so that, for each use of that component, its structure and behavior can be changed by generic values. In general, they are used to construct parameterized hardware components and can be of any type. Generic is a great asset when the design has slight changes at many places, change in the register sizes, input sizes, etc. On this work there are two main generics: G_NUM_BITS_CONCATENATE (number of bits necessary to contain the value of the correlation, intimate connected with the burst length) and G_SYM_WIDTH (bit width of the symbols). If the design is very unique then, there is no need to have generic parameters.

Among the difficulties of translating software to VHDL, the completely different architecture brings different ideas regarding data structure, organization of the code and how to behave with pipelined structures. Small differences on the result values between software and VHDL are expected due to the rounding techniques – rounding occurs when a more precise number (i.e. more fractional bits) with a less precise number (i.e. fewer fractional bits) is wanted to approximate. On this work, the technique adopted was truncation, also known by chopping. It is basically just the discard of a number of less significant bits – since the number of bits gets really large it is possible to “throw away” some of the bits and work with the rest of the bits without losing much on the quality.

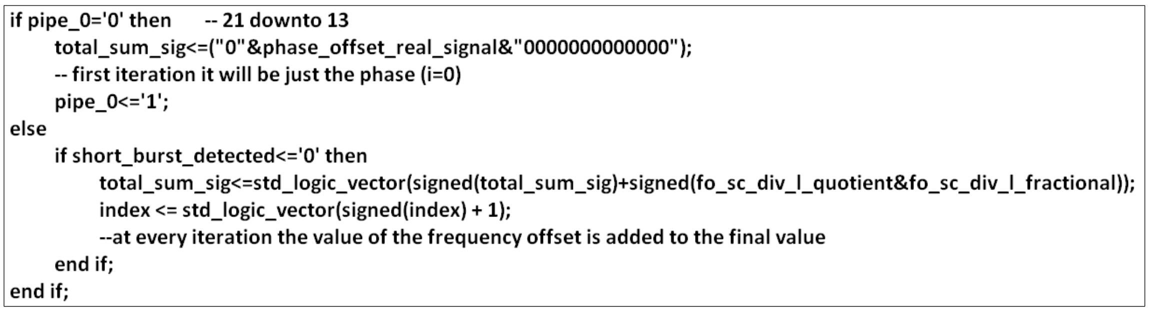
Truncation is performed on the Calc_estimate_phase_offset module. However, it is not just a simple truncation; it uses a multiplexer to define when to truncate or not. Truncation will not be done when working with small bursts – it means that the correlation value will not need so many bits for its representation and therefore the truncation would discard the value itself. Several tests and comparisons between the VHDL and the software were done in order to have a good definition of what is a small burst or not. Excerpt of the truncation code is illustrated in figure 4.20, where it can be seen that when there is a short burst detected the multiplication will take into account the whole bit width; if the short burst was not detected the bit width goes from “22 downto 5”: throwing away the 5 (“4 downto 0”) less significant bits of the symbol.



```
---- TRUNCATING WITH MULTIPLEXER
if short_burst_detected = '0' then
     ac_mult<=(signed(phase_offset_estimation_0_real(((G_SYM_WIDTH-1)*2)+...
     bd_mult<=(signed(phase_offset_estimation_0_imag(((G_SYM_WIDTH-1)*2)+...
     bc_mult<=(signed(phase_offset_estimation_0_imag(((G_SYM_WIDTH-1)*2)+...
     ad_mult<=(signed(phase_offset_estimation_0_real(((G_SYM_WIDTH-1)*2)+...
else
     ac_mult<=(signed(phase_offset_estimation_0_real(22 downto 5))*...
     bd_mult<=(signed(phase_offset_estimation_0_imag(22 downto 5))*...
     bc_mult<=(signed(phase_offset_estimation_0_imag(22 downto 5))*...
     ad_mult<=(signed(phase_offset_estimation_0_real(22 downto 5))*...
end if;
---- TRUNCATING WITH MULTIPLEXER
```

Figure 4.20: Excerpt of the VHDL code responsible for truncation of the correlation

Another implementation detail that is worth being mentioned is the relation between number of bits from the phase and frequency offset coming from the module. The frequency offset is a result of a division; therefore, it comes with an integer and a fractional part. Hence, the phase offset must be transformed into the same pattern as the frequency – the VHDL operation responsible for this can be seen in figure 4.21 – and after that at the frequency offset value will be added.

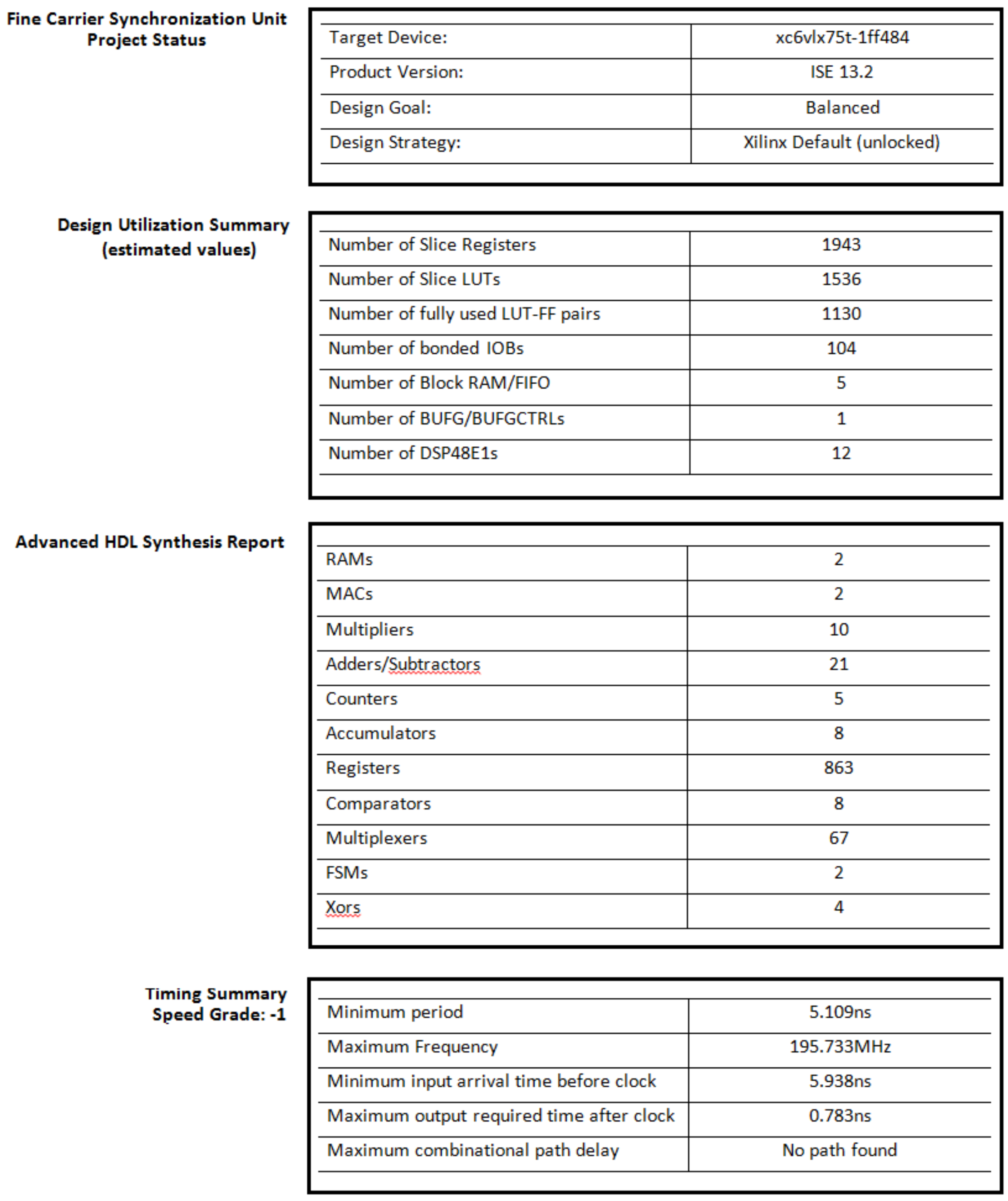


```
if pipe_0='0' then      -- 21 downto 13
     total_sum_sig<=("0"&phase_offset_real_signal&"0000000000000");
     -- first iteration it will be just the phase (i=0)
     pipe_0<='1';
else
     if short_burst_detected<='0' then
          total_sum_sig<=std_logic_vector(signed(total_sum_sig)+signed(fo_sc_div_l_quotient&fo_sc_div_l_fractional));
          index <= std_logic_vector(signed(index) + 1);
          --at every iteration the value of the frequency offset is added to the final value
     end if;
end if;
```

Figure 4.21: Excerpt of the VHDL code responsible for the adjustment of the number of bits between phase and frequency offset

4.2.4 Framework Text Report

Xilinx ISE Design Suite provides at the end of the Synthesis process a wide and complete text report about several characteristics of the several procedures done during the process. The comprehension and analysis of the quality of the VHDL code and its synthesis must go through the examination of a set of the tests given on the figure 4.22.



Fine Carrier Synchronization Unit Project Status

Target Device:	xc6vlx75t-1ff484
Product Version:	ISE 13.2
Design Goal:	Balanced
Design Strategy:	Xilinx Default (unlocked)

Design Utilization Summary (estimated values)

Number of Slice Registers	1943
Number of Slice LUTs	1536
Number of fully used LUT-FF pairs	1130
Number of bonded IOBs	104
Number of Block RAM/FIFO	5
Number of BUFG/BUFGCTRLs	1
Number of DSP48E1s	12

Advanced HDL Synthesis Report

RAMs	2
MACs	2
Multipliers	10
Adders/Subtractors	21
Counters	5
Accumulators	8
Registers	863
Comparators	8
Multiplexers	67
FSMs	2
Xors	4

Timing Summary Speed Grade: -1

Minimum period	5.109ns
Maximum Frequency	195.733MHz
Minimum input arrival time before clock	5.938ns
Maximum output required time after clock	0.783ns
Maximum combinational path delay	No path found

Figure 4.22: Text Report from Xilinx Design Tool

It is important to analyze that the implementation of the Fine Carrier Synchronization Unit when compared to the resources available on the specific target is using an negligible amount of them – for example 4% of number of DSP48E1's, 2% of the number of Slice Registers and 3% of the number of Slice LUTS. On figure 4.22 is given the exact number of components that it needs to be assembled. It is possible to realize that the developed VHDL could really become an equipment to be used in the communications systems nowadays.

By analyzing the timing summary, it is possible to conclude that inside the design there are clocked processes implemented with flip-flops – this explains why there is no combinational path founds, since in the design there are no path that go from an input pin to the output pin without going through a flip-flop; therefore, there is no maximum combinational path. It is necessary to keep in mind that every flip-flop has an input setup time, meaning that the D input must be stable this long before the active (usually rising) edge of the clock. It also has a clock to output delay measured from the rising edge of the clock to the Q output. This is useful when trying to understand the meaning of Minimum input arrival time before clock and Maximum output required time after clock – any input must be stable at least 5.938ns before the active edge of the clock (as it enters the FPGA) and that outputs will be valid no more than 0.783ns after the active edge of the clock. The minimum clock period – that must not be shorter than 5.109ns – is calculated using the worst case (longest) path from an internal flip-flop Q output to an internal flip-flop D input plus the setup and clock to output times.

4.3 General Considerations

Chapter 4 was written with the purpose of providing information about the implementation details both in C++ and VHDL; detailed explanation on how several tasks were performed and several particularities between the implementation of the same unit designed using different tools were shown. Also, it is important to take into account the mathematical background exposed throughout this chapter; which works as a ground, supporting the whole theoretical background of the synchronization system.

Section 4.1 and 4.4 are interesting to compare with the idea in mind that they both implement actually the exact same functional unit; therefore, it is possible to improve the quality of the work by comparing values of signals and variables and the output results. The comparison was done at several moments during the development – especially during the truncation implementation, it was a key to the success to assure that this was not leading to any mistake.

The next step – which will be the subject of chapter 5 - is the validation of the developed work. Validation means to submit both implementations under a wide and exhaustive set of tests and compare its results. This is necessary in order to prove that they fit to the requirements of functionality and performance of acceptable software's and VHDL's developed nowadays.

5 VALIDATION

This chapter describes the tests that were made over the implementation developed through this work. The tests have the objective to validate the mathematical background as well as proving that the Fine Carrier Synchronization Unit is working on both implementations – C++ and VHDL.

5.1 Test Environment

The system used to perform validations was a Intel Core i5 520M 2,5GHz, 4GB DDR3 533MHz. It was used two different operating systems during development, debugging and tests. The C++ software implementation, compilation and analysis were done using a Linux Ubuntu release 10.10 with kernel 2.6.35.28 and GNU Compiler Collection (gcc) version 4.4.5 [GCC, 2012]. The hardware was developed, compiled, synthesized and simulated with Xilinx ISE Design Suite 13.2 running on Windows 7 Home Premium Service Pack 1 (64-bit operating system).

5.2 Methodology

Since it would be impractical to perform validation for all the possible knowledge conditions (the number of possible and thinkable conditions is close to infinite), it was chosen to simulate conditions where there was something extreme – where it is possible to believe that something was leaning to go wrong, i.e. invalid inputs and boundaries conditions.

The entry criteria for the validation are to make scenarios to identify failures whose removal raises the software quality by increasing the reliability [PATTON, 2005]. Which means validating the software and hardware by testing them with several different kinds of inputs: at boundaries conditions (limits) and with invalid entries.

In a simplified view, the testing is basically the introduction of a noise – which is a phase or/and frequency offset – into the original signal and verifying the correction done on the other end of the chain. Figure 5.1 illustrates this flow.

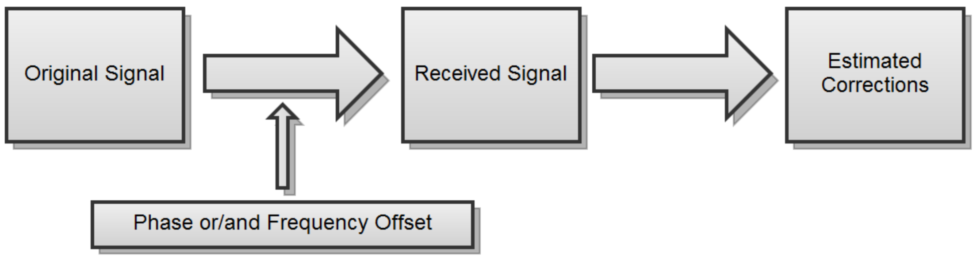


Figure 5.1: Original signal until estimated corrections flow

Once the estimated corrections are available, it is time to verify if the received signal when adopting such estimation approximates itself from the original signal. Figure 5.2 shows the idea to reconstruct the original signal and verify the reliability of the estimations done.

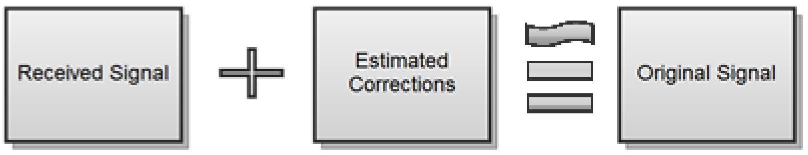


Figure 5.2: Verification of the reliability of the estimated corrections

It is important to keep in mind that all the tests, validation methods and criteria adopted are valid for C++ and VHDL as well. This is possible since the VHDL takes as input the received symbols (extracted from the software) which will contain several definitions and perform his work: calculating the correlation, estimating the parameters and performing the correction of these symbols – therefore, all the definitions of SNR, number of symbols, phase and frequency offsets are done in C++ but are also the same for the VHDL.

The methods and criteria of testing are traditionally divided into structural and functional aspects. Structural testing criteria, i.e. criteria which take into account an internal structure of the program, are in turn divided into data-flow and control-flow criteria. Data-flow criteria are based on the investigation of the ways in which values are associated with variables and how these associations can affect the execution of the program. Control-flow criteria examine logical expressions, which determine the branch and loop structure of the program.

In order to have a perfect and consistent analysis of the developed software and VHDL, it is mandatory that the noise which will be introduced is correctly configured and smartly chosen. It directly limits the detection and processing of all information. The noises discussed at this moment are the one introduced by the new module in software, called Add Offset Module, and the one from the Channel_AWGN module; it is known that the AWGN noise refers to the fact that noise eventually combines with the desired signal and is a major limiting factor in the transmission of information.

The choice of the noise that was introduced is based on the power efficiency (depending on the SNR for a specific error probability); on the bandwidth efficiency (the data rate per unit bandwidth); and implementation cost and complexity.

The pillars to the introduction of errors on the system are basically the parameters of the modules in charge of adding the noise: Add Offset Module and Channel_AWGN.

The Add Offset Module has two parameters: phase offset and frequency offset. It is also important to remind that another possible variation is the burst length and bit width of the symbols, which can be defined directly on the instantiation of the Source_bits Module on the XML input file – figure 5.3.

```
    <module>
        <instance_name>Source_Bits</instance_name>
        <start_seed>123123</start_seed>
        <num_bits>112</num_bits>
<!-- Number of bits to generate per block. Using QPSK = twice as much as symbols -->
    </module>
```

Figure 5.3: Source_bits Module instantiation and parameters definition

The burst length is defined by the parameter “num_bits” on the Source_bits Module. Since QPSK is being used, the burst length corresponds to half of the number of bits. For example, by defining the num_bits equal to 112; the burst length is, therefore, 56. The valid burst length range for this communication pattern are 56, for minimal burst length, and 2592 for maximal burst length; which implies that the minimal number of bits is 112 and 5184, respectively. These values for minimum/maximum burst length are defined by the FEC frame size maximum code rate, details can be found on [ETSI-REFERENCE, 2012]

It is important at this moment, to remind that there was defined a threshold value for bursts with length bellow and over 256 symbols. Bursts that contain from 56 to 256 symbols are considered small bursts – this will imply in a truncation over the value of the correlation (already discussed before on this work). The threshold value is important to be well defined and it is one point where great time and effort must be spent in order to guarantee a correct functionality – this is done applying several boundaries tests and comparing results with software and VHDL implementation. The detection of small bursts is shown in figure 5.4.

```
if (received_symb_le  = '1') then
    rec_symb_le_p2<= '1';
    if index_half < "0000010000000" then  --index_half <= 256 => short burst detected!
        short_burst_detected<='1';
    end if;
end if;
```

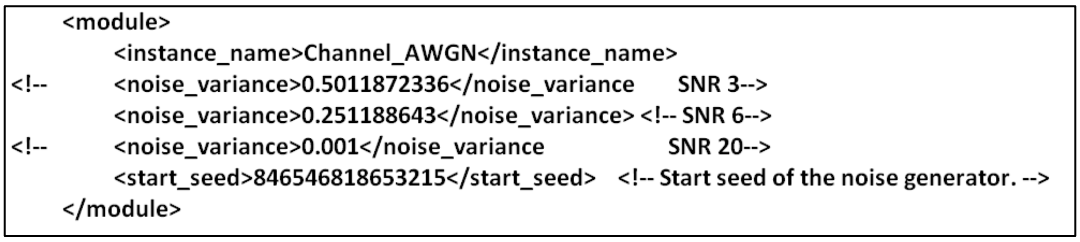
Figure 5.4: Excerpt of the code responsible for the detection of small bursts

The instantiation of the Add Offset Module and its parameters (included on the XML input file) can be seen in figure 5.5.

```
    <module>
        <instance_name>Offset</instance_name>
        <phase_offset>0.00</phase_offset>               <!-- [0..phase offset..2PI]  0 degree-->
<!--    <phase_offset>0.523598777</phase_offset>        <!-- [0..phase offset..2PI] 30 degree-->
        <frequency_offset>0.0000</frequency_offset>  <!-- [-0.5..frequency offset..0.5]-->
<!--    <frequency_offset>0.0085</frequency_offset>  <!-- [-0.5..frequency offset..0.5]-->
    </module>
```

Figure 5.5: Add Offset Module instantiation and parameters definition

The Channel AWGN Module also has heavy influence on the final result of the noise added to the communication system. The SNR can be set with its parameters. This can be seen on figure 5.6.



```
     <module>
          <instance_name>Channel_AWGN</instance_name>
<!--      <noise_variance>0.5011872336</noise_variance      SNR 3-->
          <noise_variance>0.251188643</noise_variance> <!-- SNR 6-->
<!--      <noise_variance>0.001</noise_variance              SNR 20-->
          <start_seed>846546818653215</start_seed>   <!-- Start seed of the noise generator. -->
     </module>
```

Figure 5.6: Channel AWGN Module instantiation and parameters definition

The definition of the SNR on the Channel AWGN is possible through the definition of the noise variance of the channel. For example, a noise variance equals to 0.251188643 defines a SNR equals to 6dB. The relation between noise variance and SNR is defined on equation 5.1.

$$Noise\ Variance = \frac{1}{10^{\frac{SNR}{10}}} \qquad 5.1$$

With the complete understanding of how the noise is introduced and how parameters impact the simulation tests scenarios, that are believed to comprehend all the important test cases, were defined. It is important to remember that the number of bursts can be easily defined on the XML input file – it is clear that the number of bursts will have a great impact on every statistic calculation (as more bursts are available a better approximation can be done).

The scenarios chosen – and shown in table 5.1 – are basically trying to embrace the whole implementation and it is done by parts. The first scenario only varies the phase offset. The second scenario only varies the frequency offset. The third, and last, scenario varies phase and frequency offset.

Table 5.1: Scenarios used to validation of the work

Scenarios	*Variation*	*Range*
1st	Phase Only	[-π,+π]
2nd	Frequency Only	[-0.5,+0.5]
3rd	Phase and Frequency	[-π,+π] and [-0.5,+0.5]

The SNR is constantly set to be 20dB during the three scenarios in order to mitigate the influence on the simulation of the noise from the Chanel_AWGN module and obtain an error-free transmission. With this SNR value the transmission happens practically on an ideal channel and makes possible the evaluation the errors of phase and frequency offsets introduced by the Add Offset module.

5.3 Fine Carrier Synchronization Software Analysis

The first unit test was chosen to be the verification of the calculation of the correlation value; it is the first step of the whole Fine Carrier Synchronization Unit process, therefore, it is mandatory that it is correct – once this is satisfied, it is possible to proceed with the verification of the further modules.

To verify the quality of the error estimation, it is possible to use just one value within the possible range of variation. If the module does the estimation correctly to this value, it will be correctly done to all the values in the range; therefore, there is no reason to repeat to n values. The values, which were chosen randomly, are 30° for the phase offset and $0.0044629\ radians$ for the frequency offset.

The first scenario is intended to check the addition, estimation and correction of the phase offset only. With this approach it is possible to analyze and understand exactly how and if the addition and estimation of the phase offset is acting on the system. Figure 5.7 is the example of a graphic representation of an introduction of a phase offset equal to 30° – it is possible to examine perfectly that the symbol suffered a rotation of approximately 30° counter-clockwise. The original symbol is represented by "Y" (with the coordinates 45, 45) while the received symbol by "X" (with the coordinates 15, 66).

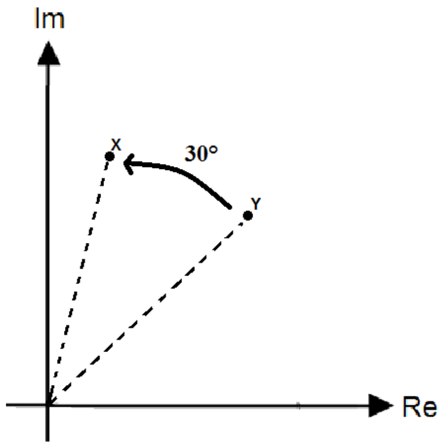


Figure 5.7: Effect of the addition of phase offset equals 30° on the symbol

It is interesting to analyze the XML output file – it contains the estimation offset values, average of the estimation, mean and variance. The important values are shown in table 5.2.

Table 5.2: Phase offset estimation related values extracted from the XML output file

	Estimated Values	*Expected Values*
Phase Offset Estimation Average (°)	30.3255	30
Variance (°)^2	9.52743e-06	4.04695e-05
Standard Deviation (°)	0.00308665	0.00636156

The next step is to check the correction of the symbols, the estimated offset is now used to perform the correction of such values – on the way to make the understanding easier; figure 5.8 contains now the corrected symbol "Z" as well as the original and the received symbol. Important to keep in mind that figure 5.8 contains an example – this possibly varies if a different symbol were taken.

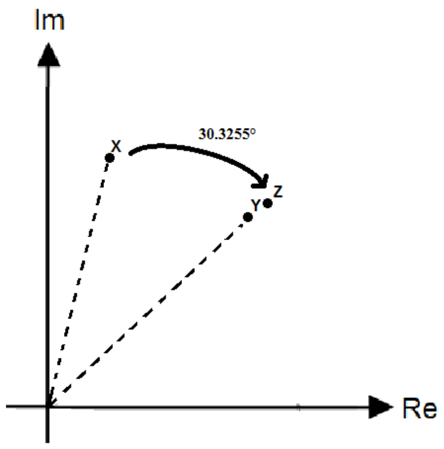


Figure 5.8: Effect of the correction of the symbol with the estimated phase offset.

Table 5.3 comes with the purpose to give the exact symbol values to the figure 5.8 and exemplifies the process that the symbols are suffering – note that this is an example.

Table 5.3: Symbol transformation process with phase offset only (software)

Symbol Position on the burst	*Original Value*	*Received Value*	*Corrected Value*
1st	(45,45)	(15,66)	(46,49)
2nd	(45,45)	(18,60)	(45,42)
3rd	(45,45)	(16,60)	(44,43)
...	...	...	…
54th	(45,45)	(17,61)	(45,44)
55th	(45,45)	(17,62)	(46,44)
56th	(45,45)	(17,61)	(45,44)

The second scenario proposes the verification of the introduction, estimation and correction of the frequency offset only. This is done following the same idea – to have a complete idea of what is going on when trying to add, estimate and correct itself, without any other external influence. During the second test scenario the frequency offset is set to be 0.00446429 radians. This value represents the maximal frequency offset possible in a 56 symbol's burst, respecting the maximal frequency possible to this burst and allowing a good visualization of the frequency offset effect on the symbols.

Figure 5.9 shows the original symbols position – represented by "Y" – and the received position – represented by "X". All the original symbols from this burst are on the "Y" position (there are 56 symbols on the exact position represented by "Y"). Due to the frequency offset they progressively "slide" through the complex plane and appear in different places.

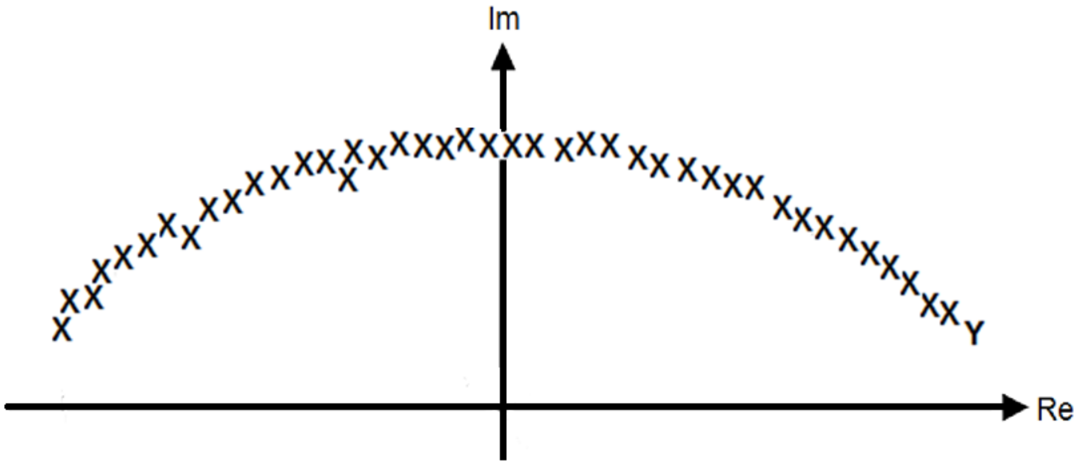


Figure 5.9: Effect of the addition of frequency offset equals 0.00446429radians on the symbol

With the proper estimation and correction, the symbols are brought back to a position more approximated to the original symbols “Y” (leading to a correct interpretation of the same) – the positions of the corrected symbols are represented by “Z”. As can be seen on the figure 5.10, the positions of “Z” are the possible position for the 56 received symbols after the correction – the symbols that were previously scattered in positions represented by “X”, now are concentrated in the positions represented by “Z”. The correction Figure 5.10 contains now the corrected symbols (“Z”) as well as the original (“Y”) and the received symbols (“X”).

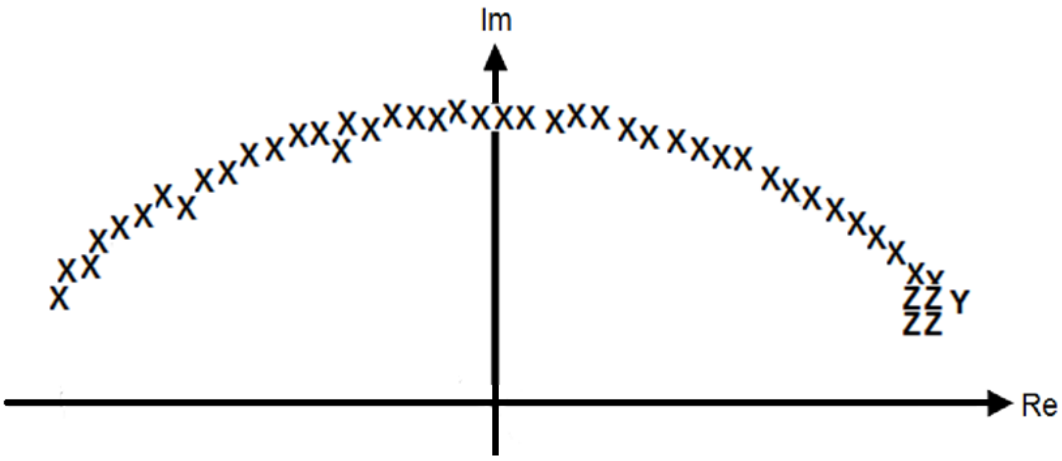


Figure 5.10: Effect of the correction of the symbols with the estimated frequency offset.

It is interesting to analyze the XML output file – it contains the estimation offset values, average of the estimation, mean and variance. The important values are shown in table 5.4 – the average was calculated from a 100000 bursts.

Table 5.4: Frequency offset estimation related values extracted from the XML output file

	Estimated Values	*Expected Values*
Frequency Offset Estimation Average (rad)	0.00441108	0.00446429
Variance (rad²)	1.94578e-05	1.99255e-05
Standard Deviation (rad)	0.0044111	0.0044638

The same approach done with the phase offset only is now done with the frequency offset. The transformation that the symbols are suffering is explicit on table 5.5 – it was taken the first three and the last three symbols as an example.

Table 5.5: Symbol transformation process with frequency offset only (software)

Symbol Position on the burst	*Original Value*	*Received Value*	*Corrected Value*
1st	(45,45)	(43,50)	(43,50)
2nd	(45,45)	(46,45)	(47,43)
3rd	(45,45)	(42,46)	(44,43)
…	..	..	..
54th	(45,45)	(-42,47)	(43,46)
55th	(45,45)	(-41,45)	(42,43)
56th	(45,45)	(-44,44)	(42,45)

The third scenario proposes the mutual effects of testing phase and frequency offset together. The same values proposed separately are now combined, keeping in mind that these values are sample values, it is possible to take any value within the acceptable range. Table 5.6 shows the transformation that the symbols are suffering when the simulation with phase equals to 30° and frequency offset equals to 0.00446429 radians was executed. Table 5.7 brings the statistics data regarding that simulation – which are found on the XML output file.

Table 5.6: Symbol transformation process with phase and frequency offset (software)

Symbol Position on the burst	*Original Value*	*Received Value*	*Corrected Value*
1st	(45,45)	(15,66)	(45,50)
2nd	(45,45)	(17,61)	(45,43)
3rd	(45,45)	(13,61)	(43,44)
…	…	…	…
54th	(45,45)	(-61,19)	(43,47)
55th	(45,45)	(-59,17)	(41,45)
56th	(45,45)	(-62,16)	(43,46)

Table 5.7: Statistics values from the phase and frequency simulation extracted from XML output file

	Estimated Values	*Expected Values*
Phase Offset Estimation Average (°)	29.0027	30
Phase Offset Variance $(°)^2$	0.256225	0.274141
Phase Offset Standard Deviation (°)	0.506187	0.523585
Frequency Offset Estimation Average (rad)	0.00442111	0.00446429
Frequency Offset Variance (rad^2)	1.96029e-05	1.99834e-05
Frequency Offset Standard Deviation (rad)	0.00442751	0.00447027

5.4 Fine Carrier Synchronization VHDL Analysis

It is important to keep in mind that the evaluation on VHDL is way more complicated than the one in software – all the statistics calculations done by the Statistics module are not performed in VHDL which implies in the manual calculation. Due to simplicity purposes, as well as during the software validation, first only the phase offset will be checked and then only frequency offset – it is important to mention that the SNR is 20dB during the VHDL evaluation.

Before testing the VHDL under the three scenarios already described, the verification of the calculation of the correlation is done and shown with table 5.8. The purpose of Table 5.8 is to verify if the manual complex multiplication is being done properly in VHDL – software and VHDL values are putted side-by-side. Table 5.9 brings the comparison between the correlation average correlation value for 10 bursts – due to timing and difficulties of providing inputs and analyzing the outputs of the VHDL the number of bursts is considerably smaller if compared to software.

Table 5.8: Comparison between the correlation values from the first burst

	VHDL Values	*Software Values*
Correlation First Half	(96975,56925)	(96975,56925)
Correlation Second Half	(96840,56970)	(96840,56970)
Correlation Total	(193815, 113895)	(193815, 113895)

Table 5.9: Average correlation value comparison

	VHDL Value (10 bursts)	*Software Value (100000 bursts)*
Correlation Average	(194052,116096)	(195005,114037)

The phase offset – estimated and expected – are shown on table 5.10 and a sample with some symbols and the process that they suffered is shown on Table 5.11. It is possible to analyze perfectly how the correction acts on the received values – during this simulation the frequency offset is defined to be zero.

Table 5.10: Estimated and expected phase offset (VHDL)

	Estimated Value	*Expected Value*
Phase offset (°)	30,234375	30

Table 5.11: Symbol transformation process with phase offset only for the first symbols on the burst (VHDL)

Symbol Position on the burst	*Original Value*	*Received Value*	*Corrected Value*
1st	(45,45)	(15,66)	(46,50)
2nd	(45,45)	(18,60)	(46,43)
3rd	(45,45)	(16,60)	(44,44)
...	...	...	…
54th	(45,45)	(17,61)	(45,44)
55th	(45,45)	(17,62)	(46,45)
56th	(45,45)	(17,61)	(45,44)

Table 5.12 shows the frequency offset – estimated and expected – regarding the simulation with frequency offset only – the phase offset was defined to be zero in order

to verify the correct functionality of the frequency correction. Table 5.13 brings the symbols from the simulation with frequency offset equals to 0.00446429 radians. The analysis of table 5.13 helps understanding the exact transformation that every symbol suffers – first the symbol's original value, then the received value and finally its corrected value. The objective of this analysis is to get a better idea of the process that happens to the symbols and its mutation values during the process that occurs inside the Fine Carrier Synchronization Unit.

Table 5.12: Estimated and expected frequency offset (VHDL)

	Estimated Value	*Expected Value*
Frequency offset (rad)	0,0044060733	0.00446429

Table 5.13: Symbol transformation process with frequency offset only (VHDL)

Symbol Position on the burst	*Original Value*	*Received Value*	*Corrected Value*
1st	(45,45)	(43,50)	(43,50)
2nd	(45,45)	(46,45)	(47,44)
3rd	(45,45)	(42,46)	(44,44)
...	…	...	…
54th	(45,45)	(-42,47)	(43,46)
55th	(45,45)	(-41,45)	(42,44)
56th	(45,45)	(-44,44)	(42,46)

Table 5.14 shows the estimated offsets next to the expected offset values. Table 5.15 contains the results of the combination of phase offset equals to 30° and frequency offset equals to 0.00446429radians.

Table 5.14: Estimated and expected phase and frequency offset (VHDL)

	Estimated Value	*Expected Value*
Phase offset (°)	28,828125	30
Frequency offset (rad)	0,0044060733	0.00446429

Table 5.15: Symbol transformation process with phase and frequency offset (VHDL)

Symbol Position on the burst	*Original Value*	*Received Value*	*Corrected Value*
1st	(45,45)	(15,66)	(45,50)
2nd	(45,45)	(17,61)	(45,44)
3rd	(45,45)	(13,61)	(43,45)
...	…	…	…
54th	(45,45)	(-61,19)	(42,48)
55th	(45,45)	(-59,17)	(41,46)
56th	(45,45)	(-62,16)	(42,48)

5.5 Software and VHDL Tests

The implementation in VHDL introduces rounding errors according to the way that the module does the truncation of the values. The objective of this section is to verify the impact of this rounding error on the VHDL results. With this goal, it was executed simulations in software (CSE) and the software results were compared with the VHDL results.

Several burst lengths (63, 64, 127, 128, 255 and 256) were tested with different average symbol values (30, 45 and 255) to verify which value would fit to the system needs and not result in errors. The values tested to become the truncation threshold were 64, 128 and 256. On table 5.18 exposes the reasons which made it impossible the use of the values 64 and 128 – several others problems were found with different configurations, the one disposed on table 5.18 act as an example. With truncation threshold defined as 256 symbols there were no error found.

Table 5.16: Analysis with the truncation threshold define as 256 symbols

	Not truncating	*Truncating with a multiplexer*	*Coarse Truncating*
Number of symbols	128	128	128
Received Symbols average	|30|	|30|	|30|
Reference Symbols Average	|30|	|30|	|30|
Multiplication of the correlation values (real part)	37725750000	8975	8975
Multiplication of the correlation values (imaginary part)	-3280500000	-760	-760
Estimated Final offset (°)	-21.09375	-23.9	-23.9

In order to provide a better understanding and a deeper analysis of how the two implementations are working when facing different phase offsets, several different phase offsets were introduced on purpose – it is interesting analyze side-by-side the results from software and VHDL. Table 5.21 shows the estimation phase and frequency offset side-by-side – with SNR equals to 3dB. By changing the SNR value to 3dB instead of 20dB a better approximation with real world is achieved – since there is more noise coming from the environment.

Table 5.17: Estimation of phase and frequency offsets side-by-side

phase offset input	*software phase offset (°)*	*software frequency offset*	*vhdl phase offset (°)*	*vhdl frequency offset*
0°	0.00278	5.43432e-06	0°	0
5°	5.08928	-8.49855e-06	4,921875°	0
10°	10.1226	-9.65789e-06	9,140625°	0
15°	15.1694	-1.07359e-05	14,765625°	0
20°	20.2232	-1.17073e-05	21,796875°	0
60°	59.6884	-1.52645e-05	61,171875°	0
90°	90.0924	-1.32207e-05	92,109375°	0
135°	135.002	-2.86087e-06	131,484375°	0

5.6 General Considerations

In general, the validation identifies failures whose removal rises the software quality by increasing the software's and VHDL's potential reliability. The testing is the measurement of the quality provided by the software and VHDL – it is possible to analyze if the original idea was achieved and/or possible.

By performing several tests on the software and VHDL and, after that, analyzing the results, it is possible to realize that the development and the whole motivation for the creation of a Fine Carrier Synchronization Unit was based on grounded studies that provide a real and concrete subject of study to nowadays communication systems.

Once the validation of the developed software and VHDL is finished there must be a critical look at the results, an analysis to check if the final result has fulfilled the expectations and if there are any point where the performance can be improved – this is the topic of the chapter 6.

6 CONCLUSION

This work provides both software and VHDL implementation of a Fine Carrier Synchronization Unit which has the purpose of increasing the quality of turbo synchronization systems. The Fine Carrier Synchronization Unit is intended to raise the quality of turbo synchronization systems by increasing its accuracy and the precision - as nowadays communication systems require. It is also an important goal to implement a dynamic and useful tool which will be ready to be introduced into any production chain or work as an academic didactic tool for other students to help the understanding of synchronization systems. Besides, it must be also mentioned here that by implementing the Fine Carrier Synchronization the approach of the CSE for new modules is validated.

The implementation of the Fine Carrier Synchronization Unit in software and the integration of the new modules into the already existing Creonic Simulation Environment is the first step successfully done. As would be expected, the first task of this work requested more effort and also more implementation time if compared with other parts of this project; a whole study about the simulation environment was necessary, which is the base for the future work as well as for a better understanding and comprehension of the basic concepts. The integration and the interfaces that were about to be used and the definition of types between such different modules had been deeply studied in order to not bring any errors during the further implementation of modules and their execution.

Since the Fine Carrier Synchronization Unit has been implemented and functional in software, the focus was the development of the VHDL module, which was expected to correspond exactly and behave on the same way as the software. The VHDL implementation was positively accomplished and tricky, since several software functions (for instance, complex multiplication) do not exist or are not synthesizable when regarding hardware description languages. Therefore, if these functions are required, it is mandatory that the programmer create them with the available functions or import them from a design tool. On the other hand, since the whole idea, approach and functionality of the Fine Carrier Synchronization Unit had been already studied exhaustively when doing the software development, there was enough time to dedicate into the creation of such functions and other challenges that emerged – for example, the VHDL initialization of several signals necessary to work with one burst after another and the management of the memory to deal with several bursts.

Once the implementation of both software and VHDL are effectively finished, the new objective, as important as their implementation, is to test, analyze, compare, and measure the quality and accuracy of the results provided. Possibly at this stage of the project any problem could appear that was undistinguished until this point of the work.

Fortunately the results are positive when both implementations were hit by a series of tests – it was notable that they both have nearly the same results with the same inputs. These slight differences found are the result of approximation loss regarding VHDL precision – which are acceptable and totally not influence on the result.

The implementation of a Fine Carrier Synchronization Unit shows the possibility of the developed work to operate as an important tool aiming improvements on communication systems by introducing an accurate technique of estimation and correction of offset parameters. Besides, it functions as well to validate the work developed on [WASENMÜLLER, 2009] – the whole mathematical background necessary to the development of the Fine Carrier Synchronization Unit comes from this work.

Due to reasons, like time and purpose, there are still some points along this work where there are enhancements to be done. For instance, the truncation of the correlation value – it is a really delicate topic since it is mandatory that this happens due to the wide range that the correlation value can assume when dealing with bursts with 56 symbols or when dealing with bursts with 2592 symbols. If this is not done, it applies a low frequency due to the bit width necessary – which would classify the Fine Carrier Synchronization Unit really outside of the commercial standards of today. Another point which may be mentioned here with the potential for improvements is the interaction between software and VHDL – the test bench files for the VHDL simulation is done altogether manually and it is quite time consuming. The idea for further works is to implement an extra embedded module inside the CSE that will be responsible for the automatic creation of VHDL test bench files.

The Fine Carrier Synchronization uses the Creonic Simulation Environment as its environment to achieve such functionality – therefore, the results are the proof that the Fine Carrier Synchronization Unit is working as expected and verified by the Creonic Simulation Environment and the VHDL simulations.

The work developed throughout this project contributes to the Creonic Simulation Environment – by introducing techniques which provide a wider approach to modern communication system’s problems – and to the consolidation and validation of the theory that was the base for the implementation of it.

REFERENCES

[ALLES, 2007] Alles, Matthias, Timo Lehnigk-Emden, Uwe Wasenmüller, and Norbert Wehn. "**Implementation Issues of Turbo Synchronization with Duo-Binary Turbo Decoding.**" *Invited paper, In Proc. 18th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC).* Athens, 2007.

[BRACK, 2005a] Brack, Toren, Uwe Wasenmüller, and Norbert Wehn. "**A Configurable IP Core for Combined Blind Frequency and Phase Synchronization of MPSK Bursts**." *In Proc. 14th IST Mobile and Wireless Communications Summit.* Dresden, 2005.

[BRACK, 2005b] Brack, Toren, Uwe Wasenmüller, Daniel Schmidt, and Norbert Wehn. "**Design Space Exploration for Frequency Synchronization of BPSK/QPSK Bursts.**" *Advances in Radio Science*, 2005.

[CLARK, 1981] Clark Jr., George C. , and J. Bibb Cain. ***Error-Correction Coding for Digital Communications (Applications of Communications Theory)***. 1981.

[CREONIC, 2012] CREONIC. <available at http://www.creonic.com>: last access: june 2012.

[DALE, 2004] Dale, Nell B. ***Programming and problem solving with C++***. Jones & Bartlett Publishers, 2004.

[DOXIGEN, 2012] DOXIGEN. <available at http://www.doxygen.org>: last access: june 2012.

[ETSI, 2012]ETSI. <available at http://www.etsi.org/WebSite/homepage.aspx>: last access: june 2012.

[ETSI-REFERENCE, 2012]ETSI Reference. <available at http://www.etsi.org/deliver/etsi_en/301700_301799/301790/01.05.01_60/en_301790v010501p.pdf>: last access: june 2012.

[GCC, 2012] *GCC.* <available at http://gcc.gnu.org/>: last access: june 2012.

[ROCHOL, 2011] Rochol, Juergen. Comunicação de dados – Série livros didáticos informática UFRGS – VOL 22 – 2011. Ed. Bookman.

[LEHNIGK-EMDEN, 2008] Lehnigk-Emden, Timo, Uwe Wasenmüller, Christina Gimmler-Dumont, and Norbert Wehn. "**Analysis of Iteration control for Turbo Decoders in Turbo Synchronization Applications**." *Advances in Radio Science* 7 (2008): 139-144.

[MENGALI, 1997] Mengali, Umberto, and Aldo N. D'Andrea. ***Synchronization Techniques for Digital Receivers (Applications of Communications Theory)***. Springer, 1997.

[MEYR, 1997] Meyr, Heinrich, Marc Moeneclaey , and Stefan A. Fechtel . ***Digital Communication Receivers, Synchronization, Channel Estimation, and Signal Processing***. Wiley-Interscience, 1997.

[MSDRG, 2012] *Microelectronic Systems Design Research Group.* <available at http://ems.eit.uni-kl.de>: last access: june 2012.

[MODELSIM, 2012] *ModelSim.* <available at http://www.mentor.com/products/fv/modelsim/>: last access: june 2012.

[NOELS, 2003] Noels, N, C Herzet, A Dejonghe, V Lottici, and H Steendam. "**Turbo Synchronization: an EM algorithm interpretation**." *IEEE Int. Conf. Communications (ICC).* Anchorage, Alaska, 2003.

[OPPENHEIM, 1989] Oppenheim, A, and R. W. Schafer. ***Discrete-Time Signal Processing***. Prentice Hall, 1989.

[PATTON, 2005] Patton, Ron. ***Software Testing (2nd Edition)***. Sams, 2005.

[RAY, 2001] Ray, Erik T. ***Learning XML***. O'Reilly Media, 2001.

[REDL, 1995] Redl , Siegmund M. . ***An Introduction to GSM***. 1st. Artech House Publishers, 1995.

[SNIFFIN, 2009] Sniffin, Robert W., and Timothy T. Pham. ***A Telemetry Data Decoding***. < available at http://deepspace.jpl.nasa.gov/dsndocs/810-005/208/208A.pdf>, last access: june 2012.

[TANENBAUM, 2003] Tanenbaum, Andrew S. ***Computer Networks.*** Prentice Hall PTR, 2003.

[TAUB, 2008] Taub, Herbert , and Donald L. Schilling. ***Principles Of Communication Systems***. McGraw-Hill Education (India) Pvt Ltd, 2008.

[TREES, 2001] Trees, Harry L. Van. ***Detection, Estimation, and Modulation Theory***. 2001.

[VHDL, 2012]VHDL. http://www.vhdl.org: last access: june 2012.

[VITERBI, 1983] Viterbi, A. J., and A. M. Viterbi. "**Nonlinear Estimation of PSK Modulated Carrier Phase with Application to Burst Digital Transmission**." *IEEE Transactions on Information Theory* 32 (1983): 543–551.

[WASENMÜLLER, 2009] Wasenmüller, Uwe, C Gimmler-Dumont, and Norbert Wehn. "**Low Complexity Synchronization Without Initial Carrier Synchronization**." *Advances in Radio Science, Volume 8*, September 2009: 123-128.

[XILINX, 2012] Xilinx. <available at http://www.xilinx.com/support/documentation/dt_ise13-2.htm>: last access: june 2012.

[XILINX CORE, 2012] *Xilinx Core Generator System.* <available at http://www.xilinx.com/tools/coregen.htm>: last access: june 2012.

[XILINX ISIM, 2009] Xilinx. *ISE ISim In-Depth Tutorial.* <available at http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ug682.pdf> : last access: june 2012.

ANNEX A <ARTICLE TG1: FINE CARRIER SYNCHRONIZATION UNIT FOR A TURBO SYNCHRONIZATION SYSTEM>

Fine Carrier Synchronization Unit for a Turbo Synchronization System

Leonardo Hax Damiani, Uwe Wasenmüller (co-advisor), Alexandre Carissimi (advisor)

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{lhdamiani,asc}@inf.ufrgs.br

wasenmueller@eit.uni-kl.de

Abstract. *The popularity of the wireless devices comes from several advantages related to this type of communication, i.e. mobility, easy installation and less cost for infrastructure. Hence it is vital to assure a reliable communication where errors can be autonomously fixed and information responsibly secured. The transmission over wireless channel results in frequency and phase offsets; additionally the received symbols are corrupted with noise. Therefore the estimation of the actual frequency and phase offset becomes a very critical task with high impact on communications performance; synchronization is a crucial part of each receiver in digital communication systems. In this context, throughout this work is proposed an implementation of a Fine Carrier Synchronization Unit that aims a better communication quality and lower its error rate.*

Resumo. *A popularidade de equipamentos sem fio decorre de uma série de vantagens relacionadas a este tipo de comunição, i.e. mobilidade, fácil instalação e menor custo para infraestrutura. Consequentemente é vital garantir-se uma comunicação confiável onde erros podem ser automaticamente corrigidos e a informação responsavelmente segura. A transmissão sobre canais sem fio resulta em deslocamentos de frequência e fase; além disso, os símbolos recebidos podem ser corrompidos com ruído. Portanto uma estimativa dos valores de deslocamento reais de frequência e fase se torna uma tarefa fundamental com grande impacto no desempenho da comunicação; sincronização é uma parte crucial em cada receptor em sistemas de comunicação digital. Nesse contexto, ao longo deste trabalho é proposto a implementação de uma Unidade de Sincronização Fina de Portadoras que visa melhorar a qualidade da comunicação e diminuir a taxa de erros da mesma.*

1 Introduction

With the increase of the mobility in our world, there is a rising necessity for people to communicate with each other and have access to information independently of the location of individuals or the information. Importance is giving by the possibility that any phone call can be essential enough to save a life, close a business deal or provide hours of leisure. Each of these examples of mobile communications proposes a challenge that can only be achieved with an efficient and reliable wireless communication.

Synchronization and channel coding/decoding are vital parts in every digital receiver for wireless communication – it reduces the errors and allows to reduce the transmit power respectively. With the increase of devices using wireless data transmission technologies, it is essential that exists efficient and responsible ways to fix errors that may happen in this kind of transmission. When using wireless channel is usual that the received data had been corrupted with some kind of noise – also timing, phase and frequency offset are introduced and somehow must be taken care of. The task of synchronization is to present data bits to the channel decoder, where the negative influences of timing, frequency and phase offset are eliminated.

In addition to detection and decoding, a receiver has also to perform synchronization i.e. to estimate a number of parameters like the carrier phase or the maximum likelihood ratio. In turbo receivers synchronization is a very challenging task – since they operate at very low signal-to-noise ratio (SNR) and therefore classical synchronizers may fail to provide reliable estimated parameters. Turbo Synchronization is the idea of taking benefits from the soft information available in turbo receivers in order to improve the quality of the estimated delivered by the synchronizer [1].

Due to reasons as time and purpose of this work, synchronization will be the main subject. Channel decoding will be left aside but it has also potential to be topic of a future work. This paper will be focused on the frequency and phase synchronization of bursts with linear modulation, i.e. Quadrature Phase Shift Keying (QPSK) modulation. Timing synchronization – i.e. the optimal sampling time is properly carried out before. The system aims the Digital Video Broadcast – Return Channel via Satellite (DVB-RCS) standard, which is an ETSI satellite communication standard [2].

It is a known fact that simulations can reduce development time and costs. A project was created at the Microelectronic Systems Design Research Group from the Technical University of Kaiserslautern, which developed the software Creonic Simulation Environment – CSE. The purpose of CSE is to allow for the integration of complex simulations environments. The simulation of the synchronization task for such communication systems has enormous importance on the whole project.

CSE will be the starting point for the work which is about to be developed. New features aiming the Fine Carrier Synchronization Unit will be developed, tested and introduced into the already existing simulation environment. Hardware implementations on VHDL of these new features are also intended. Further on this work, will be possible to evaluate both – software and hardware – according to theory of communication systems and a good statistical output will be available.

The rest of this paper is structured as follows. In section 2, a brief overview of the basic concepts needed and technical concepts involved in this work. Section 3 shows the whole functionality of CSE and advantages of taking it as the first step into the Fine Carrier Synchronization Unit. Section 4 deals with the objectives and methodology adopted, how it is going to be done the implementation, which development environments are going to be used and also how it will treat the question of evaluation of the precision. Finally, Section 5 brings the schedule related to the implementation of this work.

2 Basic Concepts

At this point, it is important to understand what exactly involves synchronizations systems, why and how errors happen. Synchronization consists of the estimation of unknown parameters of frequency and phase offset, and the removal of all possible damaging effects introduced by these parameters.

In every communication transmission there will be a mapper, which is responsible to convert the binary bit stream into modulated symbols. Modulated means that the bits are going to be organized into an alphabet, which defines how many symbols are available and how they work in this "language" that they communicate with – this alphabet must be previously defined and known by both ends. The communication can be represented in a complex plane and, for example, a Quadrature Phase Shift Keying (QPSK) has a defined alphabet with 4 symbols, equally divided on the plane. Consequently, it's natural to understand that for every 90° or π/2, will be the area where one symbol will be represented. It is usual to refer this organization of symbols in a complex plane as constellation diagram (Figure 1). There are several modulations available and in use nowadays, i.e. BPSK, QPSK, 8PSK, 16-QAM, 64-QAM, etc [3].

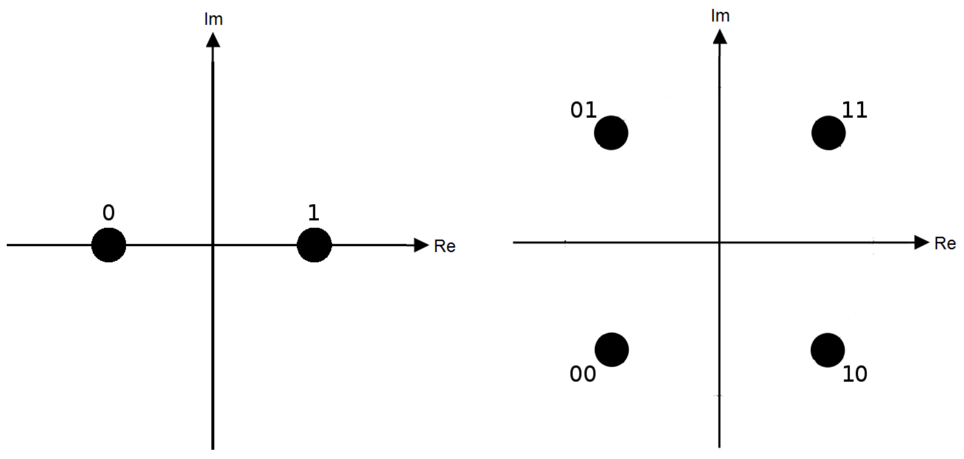


Figure 1. Constellation diagram for BPSK (left) and QPSK (right) with gray coding

The modulation would work perfectly if were not errors, noise and degradation of signal. To understand how an error occurs, it is important to note that once something went wrong these symbols are not going to be on the exact expected place. Due to the noise the symbol – the "point" in figure 1 – shifts its position in the complex plane when compared to the original position. This shifting will not always result in wrong interpretation – it can vary certain acceptable range and it will still be considered as the right symbol.

In order to explain the figure 1, it is remarkable to remind that the number of symbols per bits respects the formula 1, which represents that, for example, for every 1 bit, 2 symbols can be represented and for 2 bits, 4 symbols can be represented.

$$Number\_of\_Bits = log_2\, Number\_of\_Symbols \qquad (1)$$

The problem starts when the error is bigger than the range of acceptable variation. In a very noisy environment, for matter of explanation, figure 2 makes it more understandable.

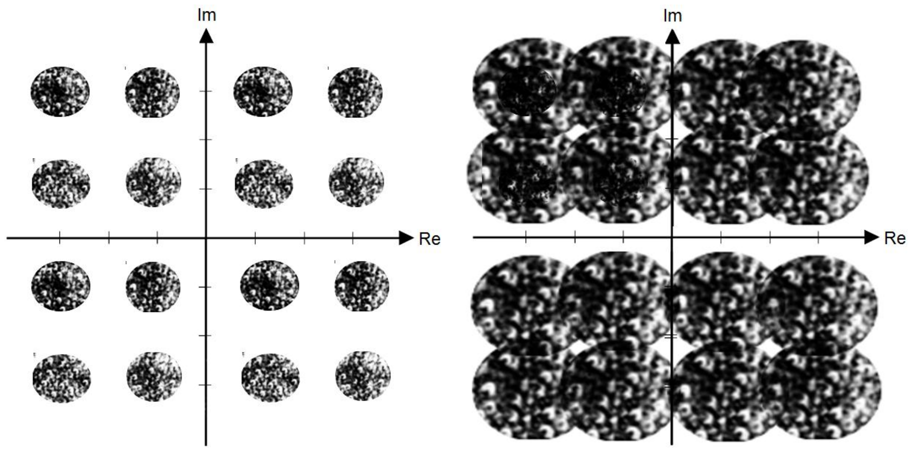


Figure 2. 16-QAM with an acceptable range variation (left – high SNR) and a very noisy range variation (right – low SNR)

Analyzing figure 2, it is easy to see that many of the symbols were read and classified wrongly since the noise was really important and there is no way to define which symbol belongs to which region on the complex plane. In figure 2, it is easy to realize that the symbols from each quadrant can suffer with high noise and transform themselves into a region where it is impossible to distinguish where it is originally from – the result is an error of the interpretation of the signal.

There is one parameter known in communication systems as Signal-to-Noise Ratio – SNR – which represents the ratio between the energy of the signal and the energy of the noise. SNR is going to define how bad the signal will get after going through the channel. It is directly influenced by the noise variation of the channel. Also included in the negative parameters of any synchronization are the frequency and phase offset.

Figure 2 can be observed also by differentiating the values of SNR: left part of figure 2 is the representation of a high SNR value; while the right part of figure 2 corresponds to a low SNR value.

Frequency offset exists as consequence from the difference between the oscillator from the transmitter (TX) and the one from the receiver (RX); oscillators from TX and RX cannot be exactly equal. In figure 3 all the symbols of a burst are represented on the same complex plane. To comprehend properly how the frequency offset can be observed, it is primordial to understand that they come not at the same exactly moment; but for teaching purposes this view of the complex plane makes it easy – figure 1 shows a QPSK example where there is no frequency offset. Frequency offset can be easily understood by taking into account the differences among figure 1 and figure 3. The frequency offset will add a constant and incrementally error to every received symbol, which is the reason why the symbols on figure 3 have a variation from the first to the last received symbol, as can be seen, for example, on symbol "10".

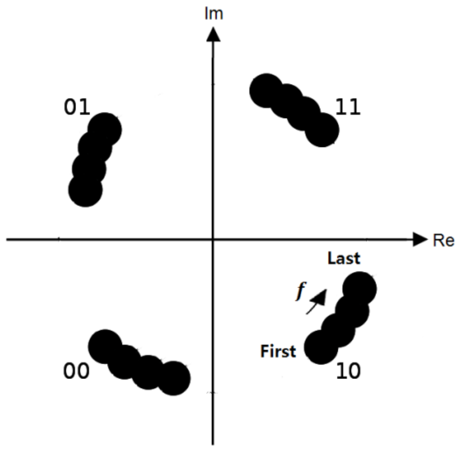


Figure 3. Illustration of a frequency offset (*f*) in a QPSK modulation

In figure 4 is shown how the phase offset acts related to the QPSK, it is a main key to understand that the phase offset is well defined to the whole bit stream – it will act in every symbol exactly in the same way. On the other hand, when related to frequency offset, it will have a higher impact as the bit stream reaches the end – the impact of the frequency offset will not be the same for every symbol.

Since the complex plane is equally divided, in QPSK, for example, by four known symbols, figure 4 show that there is no error on the interpretation of each symbol, on this case the symbols are still recognized as the expected even though there is a phase offset (α) introduced. Figure 4 comes with the purpose to show that not every phase offset would result in error.

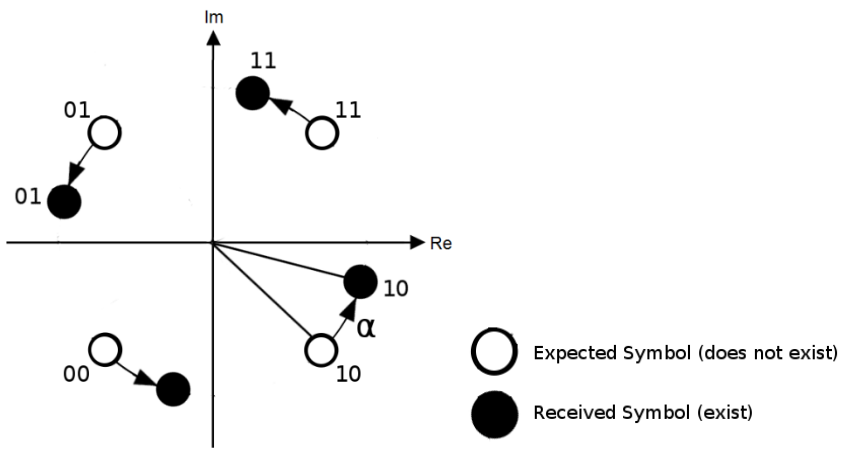


Figure 4. Illustration of an error free phase offset (α) in a QPSK modulation

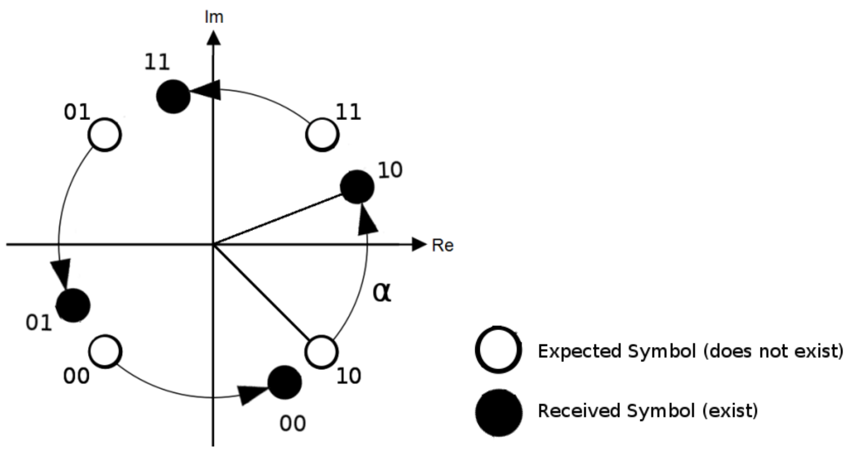


Figure 5. Illustration of an error by phase offset (α) in a QPSK modulation

On the other hand, with figure 5 it is prominent – and also emphasized on by not solid circles – that the phase offset introduced add an error to the interpretation of this communication. The symbols – represented by a solid black circle – were located in a quadrant of the complex plane and after the addition of the phase offset, they are on a different one – represented by a not solid circle. This means that the interpretation of them will result in an error, which can be corrected with a right estimation of this phase offset and the future correction of it.

3 Creonic Simulation Environment - CSE

CSE is the simulation environment that provides a set of tools and functionalities needed to simulate real world communications [4]. It offers the possibility to reduce development time and costs – a well-known problem to every project. Other goals of CSE are the ease of use, reusability and the extensibility to new applications and standards – it was designed with the goal to provide a simulation environment where users would be able to improve their experience with the software by developing new features and applications.

CSE was developed by the Microelectronic System Design Research Group especially Dr.-Ing Timo Lehnigk-Emden and Dr.-Ing Matthias Alles – both are now former researchers from the Microelectronic System Design Research Group. Projects with such delicate design and development questions must be highly detail oriented. Hence choices of design have been adopted and respected throughout the whole development of the software. As for example, was defined a fixed interface and configuration procedures for functional modules, strict coding and documentation guidelines, and also fully object oriented design. For the documentation purposes, it was used a documentation tool called Doxygen [5] – it generates automatically the documentation from a set of documented source files.

C++ was the programming language chosen to the implementation of CSE. Which is a clever option based on the design choices made by the original developers of the software. C++ contains a good and richer standard library, if compared to C, and also support to both the structured programming and object orientation. Therefore, the new modules that are about to be implemented and integrated to the system must also follow the same language. It will be used the GCC as compiler – it is the native compiler included on the GNU/Linux system.

The simulation environment is composed of functional modules; they work as versatile pieces available to be organized together following the needs of the user, i.e. noise generator or channel decoder. These functional modules are connected with each other providing the possibility to create complex simulation chains. It is important to emphasize that this functionality is only possible due to the design choices made before. This simulation environment is extremely useful for projects related to communication systems, once the simulations costs – related to configuration and connection – used to cost a lot of time, hours or days, for the developers, now can be easily finished in a few minutes.

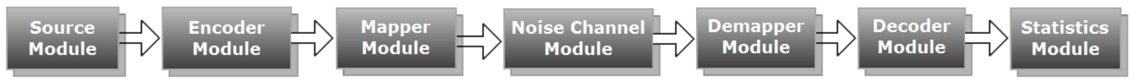


Figure 6. Basic simulation chain and functional modules connection available on CSE

In figure 6 is shown the basic simulation chain available on CSE, it represents the whole path that the bit stream will make. The source module generates random bit sequences, it is responsible for calculate one block of bits and store them into the output buffer. The Encoder module changes a signal or any other data into a code; code may optimize for purposes of compressing for transmission or storage, encrypting or adding redundancies to the input code. The Mapper module is responsible for mapping bits to modulation symbols. The Noise Channel module adds to the simulation effects of real life – the impairment to communication is a linear addition of white noise with a constant spectral density and a Gaussian distribution of amplitude. The Demapper module receives the symbols from the channel and extracts the hard bits, the LLR values (log likelihood ratio – is a statistical test used to compare the fit of two models) and the bit probabilities. The Decoder module will be in charge of the reverse operation of the encoder – changing the code into a set of signals. The Statistics module is responsible for comparing the input bits and the output bits, it takes into account the total amount of bits and how many of them are different. Besides it computes this information into different statistical parameters.

The software objective is to allow the user to create its own simulation environment and also with a rich documentation it encourages to create new functional modules and test them inside the whole system – analyzing the results and the performance of the new changes or applications. This feature of the software is a great ease towards the development of any new module, on this work is proposed the implementation of a Fine Carrier Synchronization Unit. The new module will be completely developed during this work following the method developed on [5] and it is located between the Noise Channel Module and the Demapper Module for test purposes – this means that it will be exactly the “first” part at the receiver’s end. In a real system communication it will work iteratively after the decoder. The model used to generating the noise is the Additive White Gaussian Noise – AWGN.

In this context of CSE, the Fine Carrier Synchronization is a technique that impacts the communication positively by providing ways to do it more accurately, automatically and aiming minimization of errors. With this new module it will be possible to decrease the negative influences from noise by calculating the offset parameters – frequency and phase – and correcting. The Fine Carrier Synchronization reacts to every bit stream used on the simulation environment by being located sequentially on the simulation chain – figure 7.

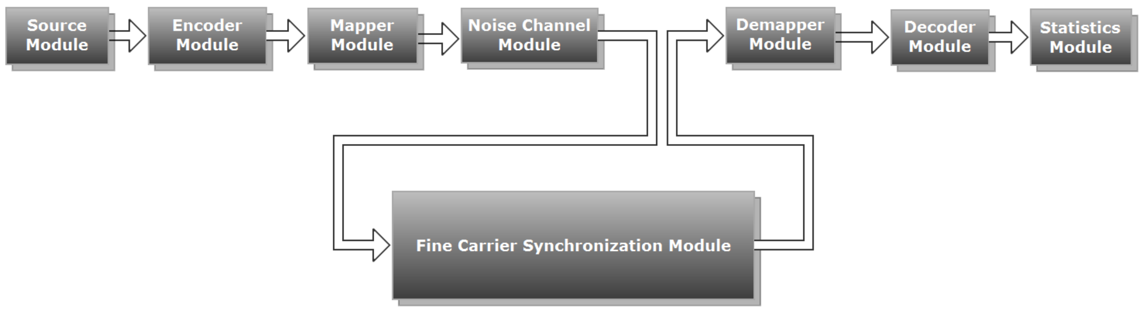


Figure 7. Fine Carrier Synchronization Unit Module position regarding the basic simulation chain

A proper implementation of the synchronization module can be simplified if the idea of how it is done is divided again into three smaller modules. As shown in figure 8, it is possible to recognize subtasks to achieve the objective of this module. By having different and identified subtasks it is obvious to create sub modules - the subtasks can be easily identified, it makes the design of logic simpler and more accurate. This way, the implemented software will respect some of its principles – reusability, flexibility and object orientation.

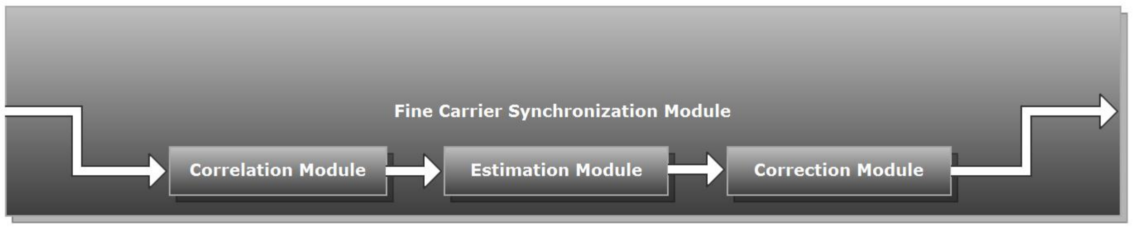


Figure 8. Fine Carrier Synchronization Unit Module modularization

The Correlation Module will be responsible for the calculation of the correlation – a statistical measurement of the relationship between the two bit streams – reference and received symbols. The reference symbols are exactly the same bit stream provided by the source module. The received symbols are bit stream that are being transmitted and had been through the others modules – encoder, mapper and noise channel. The variable that allows the execution of the correlation calculation are the bit stream – original and received. Before the Noise Channel module they are perfectly correlated, since they are exactly the same. After the addiction of the noise, they are correlated – not perfectly anymore – once the presence of certain characteristics on the original bit stream will react in order to left a sort of trace of these characteristics on the received bit stream.

As the Correlation Module produces its output, the correlation value, the Estimation module can do its part – estimate the frequency and phase offset. This is possible based on the average phase of the first and the rear part of the correlation value – when calculating the correlation value it will be divided into two parts in order to provide the values needed for the estimation of the wanted parameters.

Once the value of frequency and phase offset are estimated and available for the next module, the moment when the last task of the Fine Carrier Synchronization Module has come. The Correction module will be responsible for the correction of the noisy signal received in order to decrease the error rates.

For sake of implementation, adaptation and tests of the Fine Carrier Synchronization Module, it is needed a way to provide frequency and phase offset inputs to CSE. This offset is going to be introduced as the last step before the Noise Channel Module and its only purpose is to check if the Fine Carrier Synchronization Module is working properly. Therefore, it will be possible to define a frequency and/or phase offset input and analyze the estimation of these values and the proper correction of the symbols transmitted. The Add Offset module will model the frequency and phase offset at the transmitter side – which means that it will include an error to the bit stream.

During the development of this work, the focus is the implementation of the new modules added on figure 9 – Add Offset and Fine Carrier Synchronization. The model used to define parameters on both modules is available on [7]. It is also important to keep in mind that several changes on the Statistics module will also be done in order to embrace the new parameters and functionalities.

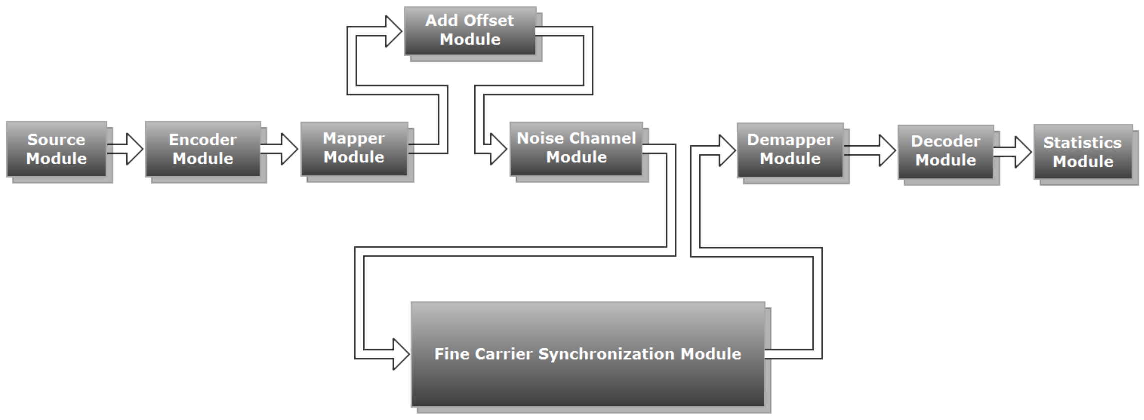


Figure 9. Add Offset and Fine Synchronization Module positions regarding the basic simulation chain

4 Objectives and Methodology

This work has as purpose two main objectives:

1. Implementation of the Fine Carrier Synchronization module in software and integrate it to the CSE, as well as the implementation of the same module in VHDL.
2. Analysis, comparison and evaluation of the accuracy of the implemented software and its correspondent in VHDL.

To achieve the first goal, it will be taken into account the whole functionality of techniques of Synchronization for digital receivers. Once comprehended deeply the behavior, it is a certain that a better and effective approach on the implementation of new synchronizations modules for any communications system will be reached [8, 9]. With the support of the CSE, is possible to focus exactly where this work proposes: Fine Carrier Synchronization Unit. It is aimed the software implementation and the integration with the CSE as a new module.

The hardware implementation, in VHDL, can be defined exclusively based on the needs of this work; it is not – so far – part of a bigger project. The digital system design tool that will be used is the Xilinx ISE Design Suite V13.2 and the chosen language is VHDL [10, 11]. It is a powerful and versatile description language, with multiple mechanisms to support design hierarchy and support for multiple levels of abstraction.

As soon as the first objective of this work is successfully accomplished, it is the moment when the accuracy needs to be verified; is essential that the output software and hardware are with a high excellence to move further with this work. There is completely no point in developing, spending time and effort to analyze a system that does not fulfill the requirements of nowadays communication systems. At this specific moment is also vital to re-examine new ways to improve processes and run them repeatedly - ensuring credibility, quality and functionality.

Hardware simulations and analysis are known by being extremely time-consuming. Taking into account the fact that the evaluation and test of it are necessary in order to have a reliable and reasonable implementation; it must be found a way to bypass this problem and prove its functionality and reliability. Therefore, the idea, to improve and optimize the simulations and analysis, is to have the support on this task with the software. The hardware and software will be implemented based on the same study; consequently they will be doing the exact same calculations in the end – of course designated to different platforms. This way, it is intended to have a higher number of cases on the software then on hardware but proceed with both evaluations together – also based on comparisons and exchange of information between the two implementations. The VHDL is intended to be simulated and synthetized with the same framework that will be used to its development.

In order to achieve a deep analysis of the developed software and hardware – regarding the Statistical module originally included on CSE – some features will be added to this module with the purpose of statistically analyze the Fine Carrier Synchronization Module and its functionality.

Among the objectives of this work is the implementation of such Fine Carrier Synchronization Unit in hardware. At this point it is crucial to understand some differences between the software to the hardware implementation, for example, there is completely no use to the implementation of the Add Offset Module. The “real world” will be in charge of this task – adding some frequency and/or phase offset to the set of bits. By developing the exactly same thing as in software, it is possible to assure that it will have the same functionality; this way, all efforts must be done on the hardware implementation of the Fine Carrier Synchronization Module. Simulations and comparisons between both software and hardware implementations will make part of the usual day-to-day while this project is under development.

5 Schedule

For the TG2, which is going to be done during the subsequent months, it was defined five activities to be developed, implemented and analyzed between March of 2012 and

July 2012. The table 1 shows exactly how the schedule is defined and how it will proceed.

Activities	March	April	May	June	July
Implementation of the Fine Carrier Synchronization Unit (Software)	**X**	**X**			
Implementation of the Fine Carrier Synchronization Unit (VHDL)		**X**	**X**		
Analysis, comparison and evaluation of the implemented software and VHDL			**X**	**X**	
Writing			**X**	**X**	
Presentation					**X**

Table 1. Schedule for the second part of this work.

6 References

1. N. Noels, C. Herzet, A. Dejonghe, V. Lottici, H. Steendam.: "Turbo synchronization: an EM algorithm interpretation". Presented at the IEEE Int. Conf. Communications (ICC), Anchorage, AL, May 2003.
2. http://www.etsi.org (accessed on 03/31/2012).
3. Tanenbaum, Andrew S.: Computer Networks (4th edition), Prentice Hall PTR, 2002.
4. www.creonic.com (accessed on 03/31/2012).
5. www.doxygen.org (accessed on 03/31/2012).
6. Wasenmüller, U, C. Gimmler-Dumont, N. Wehn..:Low Complexity synchronization without initial carrier Synchronization, in: Advances in Radio Science,Volume 8, pages 123-128, September, 2009, Miltenberg, Germany.
7. Meyr, H., Moeneclaey, M., and Fechtel, S. A.: Digital Communication Receivers, John Wiley & Sons Inc., 1998.
8. Mengali, U. and D'Andrea, A.: Synchronization Techniques for Digital Receivers, Plenum Publishing Corporation, New York, Plenum Publishing Corporation,1997.
9. Alles, M., Lehnigk-Emden, T.,Wasenmüller, U., and Wehn, N.: Implementation Issues of Turbo Synchronization with Duo-Binary Turbo Decoding, in: Proc. 19th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC) 2007, Athens, Greece, 2007.
10. http://www.xilinx.com/support/documentation/dt_ise13-2.htm (accessed on 03/31/2012).
11. http://www.vhdl.org/ (accessed on 03/31/2012).

ANNEX B < XILINX ISE DESIGN TOOL SYTHESIS REPORT>

```
Release 13.2 - xst O.61xd (nt64)
Copyright (c) 1995-2011 Xilinx, Inc.  All rights reserved.
--> Parameter TMPDIR set to xst/projnav.tmp


Total REAL time to Xst completion: 1.00 secs
Total CPU time to Xst completion: 0.67 secs


=========================================================================
*                      Synthesis Options Summary                        *
=========================================================================
---- Source Parameters
Input File Name                    : "phase_estimation.prj"
Input Format                       : mixed
Ignore Synthesis Constraint File   : NO

---- Target Parameters
Output File Name                   : "phase_estimation"
Output Format                      : NGC
Target Device                      : xc6vlx75t-1-ff484

---- Source Options
Top Module Name                    : phase_estimation
Automatic FSM Extraction           : YES
FSM Encoding Algorithm             : Auto
Safe Implementation                : No
FSM Style                          : LUT
RAM Extraction                     : Yes
RAM Style                          : Auto
ROM Extraction                     : Yes
Shift Register Extraction          : YES
ROM Style                          : Auto
Resource Sharing                   : YES
Asynchronous To Synchronous        : NO
Shift Register Minimum Size        : 2
Use DSP Block                      : Auto
Automatic Register Balancing       : No

---- Target Options
LUT Combining                      : Auto
Reduce Control Sets                : Auto
Add IO Buffers                     : YES
Global Maximum Fanout              : 100000
Add Generic Clock Buffer(BUFG)     : 32
Register Duplication               : YES
Optimize Instantiated Primitives   : NO
Use Clock Enable                   : Auto
Use Synchronous Set                : Auto
Use Synchronous Reset              : Auto
Pack IO Registers into IOBs        : Auto
Equivalent register Removal        : YES

---- General Options
Optimization Goal                  : Speed
Optimization Effort                : 1
```

```
Power Reduction                    : NO
Keep Hierarchy                     : No
Netlist Hierarchy                  : As_Optimized
RTL Output                         : Yes
Global Optimization                : AllClockNets
Read Cores                         : YES
Write Timing Constraints           : NO
Cross Clock Analysis               : NO
Hierarchy Separator                : /
Bus Delimiter                      : <>
Case Specifier                     : Maintain
Slice Utilization Ratio            : 100
BRAM Utilization Ratio             : 100
DSP48 Utilization Ratio            : 100
Auto BRAM Packing                  : NO
Slice Utilization Ratio Delta      : 5

---- Other Options
Cores Search Directories           : {"ipcore_dir"  }

=========================================================================


=========================================================================
*                          HDL Synthesis                                *
=========================================================================

Synthesizing Unit <phase_estimation>.
    Summary:
        no macro.
Unit <phase_estimation> synthesized.

Synthesizing Unit <calc_estimate_phase_offset>.
    Summary:
        inferred   8 Multiplier(s).
        inferred  19 Adder/Subtractor(s).
        inferred 660 D-type flip-flop(s).
        inferred   4 Comparator(s).
        inferred  16 Multiplexer(s).
Unit <calc_estimate_phase_offset> synthesized.

Synthesizing Unit <romshift2_1>.
    Summary:
        inferred  22 D-type flip-flop(s).
        inferred   1 Multiplexer(s).
        inferred   1 Finite State Machine(s).
Unit <romshift2_1> synthesized.

Synthesizing Unit <rom>.
    Summary:
        inferred   1 RAM(s).
        inferred  12 D-type flip-flop(s).
Unit <rom> synthesized.

Synthesizing Unit <smartscale_generic_shift_1_1>.
    Summary:
        inferred   3 Adder/Subtractor(s).
        inferred  79 D-type flip-flop(s).
        inferred   1 Comparator(s).
        inferred   6 Multiplexer(s).
Unit <smartscale_generic_shift_1_1> synthesized.

Synthesizing Unit <romshift2_2>.
    Summary:
        inferred  22 D-type flip-flop(s).
        inferred   1 Multiplexer(s).
        inferred   1 Finite State Machine(s).
Unit <romshift2_2> synthesized.

Synthesizing Unit <smartscale_generic_shift_1_2>.
    Summary:
        inferred   3 Adder/Subtractor(s).
        inferred  96 D-type flip-flop(s).
```

```
                inferred   1 Comparator(s).
                inferred   6 Multiplexer(s).
Unit <smartscale_generic_shift_1_2> synthesized.

Synthesizing Unit <frequency_phase_estimation>.
    Summary:
                inferred   1 Adder/Subtractor(s).
                inferred  37 D-type flip-flop(s).
                inferred   8 Multiplexer(s).
Unit <frequency_phase_estimation> synthesized.

Synthesizing Unit <FSM_for_fine_synchronization>.
    Summary:
                inferred   1 Adder/Subtractor(s).
                inferred   9 D-type flip-flop(s).
                inferred   2 Multiplexer(s).
Unit <FSM_for_fine_synchronization> synthesized.

Synthesizing Unit <rot_memory>.
    Summary:
                inferred   3 Adder/Subtractor(s).
                inferred  55 D-type flip-flop(s).
                inferred  11 Multiplexer(s).
Unit <rot_memory> synthesized.

Synthesizing Unit <frequency_corrector>.
    Summary:
                inferred   2 Adder/Subtractor(s).
                inferred  77 D-type flip-flop(s).
                inferred   2 Comparator(s).
                inferred   9 Multiplexer(s).
Unit <frequency_corrector> synthesized.

Synthesizing Unit <phase_corrector>.
    Summary:
                inferred   4 Multiplier(s).
                inferred   4 Adder/Subtractor(s).
                inferred 233 D-type flip-flop(s).
                inferred   2 Multiplexer(s).
Unit <phase_corrector> synthesized.

=========================================================================
HDL Synthesis Report

Macro Statistics
# RAMs                                                 : 2
 4096x9-bit single-port Read Only RAM                  : 2
# Multipliers                                          : 12
 18x18-bit multiplier                                  : 4
 9x9-bit multiplier                                    : 8
# Adders/Subtractors                                   : 36
 10-bit adder                                          : 2
 12-bit adder                                          : 1
 12-bit addsub                                         : 1
 13-bit adder                                          : 2
 13-bit addsub                                         : 1
 13-bit subtractor                                     : 2
 19-bit adder                                          : 1
 19-bit subtractor                                     : 1
 23-bit adder                                          : 1
 29-bit adder                                          : 10
 29-bit subtractor                                     : 2
 30-bit adder                                          : 2
 38-bit adder                                          : 1
 38-bit subtractor                                     : 1
 5-bit adder                                           : 1
 5-bit subtractor                                      : 1
 6-bit subtractor                                      : 1
 7-bit adder                                           : 4
 9-bit subtractor                                      : 1
# Registers                                            : 99
 1-bit register                                        : 31
 10-bit register                                       : 1
 12-bit register                                   : 6
 13-bit register                                   : 7
 18-bit register                                   : 6
 19-bit register                                   : 2
 23-bit register                                   : 2
 29-bit register                                   : 12
 30-bit register                                   : 4
 36-bit register                                   : 4
 38-bit register                                   : 4
 5-bit register                                    : 2
 6-bit register                                    : 6
 9-bit register                                    : 12
# Comparators                                      : 8
 13-bit comparator greater                         : 2
 13-bit comparator lessequal                       : 3
 13-bit comparator not equal                       : 1
 5-bit comparator greater                          : 1
 6-bit comparator greater                          : 1
# Multiplexers                                     : 62
 1-bit 2-to-1 multiplexer                          : 26
 10-bit 2-to-1 multiplexer                         : 2
 12-bit 2-to-1 multiplexer                         : 2
 13-bit 2-to-1 multiplexer                         : 8
 18-bit 2-to-1 multiplexer                         : 8
 23-bit 2-to-1 multiplexer                         : 3
 30-bit 2-to-1 multiplexer                         : 2
 38-bit 2-to-1 multiplexer                         : 2
 5-bit 2-to-1 multiplexer                          : 3
 6-bit 2-to-1 multiplexer                          : 6
# FSMs                                             : 2
# Xors                                             : 4
 1-bit xor2                                        : 4

=========================================================================
INFO:Xst:1767 - HDL ADVISOR - Resource sharing has identified that some arithmetic operations in this design can share the
same physical resources for reduced device utilization. For improved clock frequency you may try to disable resource sharing.

=========================================================================
*                       Advanced HDL Synthesis                          *
=========================================================================

Reading core <ipcore_dir/divider.ngc>.
Reading core <ipcore_dir/Rot_symbols.ngc>.
Reading core <ipcore_dir/SCL_LUT_9.ngc>.
Loading core <divider> for timing and area information for instance <part8>.
Loading core <Rot_symbols> for timing and area information for instance <part11>.
Loading core <SCL_LUT_9> for timing and area information for instance <dds>.

Synthesizing (advanced) Unit <FSM_for_fine_synchronization>.
The following registers are absorbed into counter <pipe>: 1 register on signal <pipe>.
Unit <FSM_for_fine_synchronization> synthesized (advanced).

Synthesizing (advanced) Unit <calc_estimate_phase_offset>.
The following registers are absorbed into counter <index>: 1 register on signal <index>.
The following registers are absorbed into accumulator <ac_0>: 1 register on signal <ac_0>.
The following registers are absorbed into accumulator <bd_0>: 1 register on signal <bd_0>.
The following registers are absorbed into accumulator <ad_0>: 1 register on signal <ad_0>.
The following registers are absorbed into accumulator <ac_1>: 1 register on signal <ac_1>.
The following registers are absorbed into accumulator <bc_0>: 1 register on signal <bc_0>.
The following registers are absorbed into accumulator <bd_1>: 1 register on signal <bd_1>.
The following registers are absorbed into accumulator <ad_1>: 1 register on signal <ad_1>.
The following registers are absorbed into accumulator <bc_1>: 1 register on signal <bc_1>.
Unit <calc_estimate_phase_offset> synthesized (advanced).

Synthesizing (advanced) Unit <frequency_corrector>.
The following registers are absorbed into counter <index>: 1 register on signal <index>.
Unit <frequency_corrector> synthesized (advanced).

Synthesizing (advanced) Unit <phase_corrector>.
	Multiplier <Mmult_r_im_reg[2][8]_cos_reg[8]_MuLt_2_OUT> in block <phase_corrector> and adder/subtractor
<Madd_im_temp1[17]_im_temp2_reg[17]_add_7_OUT> in block <phase_corrector> are combined into a
MAC<Maddsub_r_im_reg[2][8]_cos_reg[8]_MuLt_2_OUT>.
```

```
        The following registers are also absorbed by the MAC: <im_temp1> in block <phase_corrector>, <r_im_out_s1> in
block <phase_corrector>.
        Multiplier <Mmult_r_re_reg[2][8]_cos_reg[8]_MuLt_4_OUT> in block <phase_corrector> and adder/subtractor
<Msub_re_temp1[17]_re_temp2_reg[17]_sub_7_OUT<18:0>> in block <phase_corrector> are combined into a
MAC<Maddsub_r_re_reg[2][8]_cos_reg[8]_MuLt_4_OUT>.

        The following registers are also absorbed by the MAC: <re_temp1> in block <phase_corrector>, <r_re_out_s1> in block
<phase_corrector>.
        Found pipelined multiplier on signal <r_im_reg[1][8]_sin_value[8]_MuLt_5_OUT>:
                - 2 pipeline level(s) found in a register connected to the multiplier macro output.
                Pushing register(s) into the multiplier macro.
        Found pipelined multiplier on signal <r_re_reg[1][8]_sin_value[8]_MuLt_3_OUT>:
                - 2 pipeline level(s) found in a register connected to the multiplier macro output.
                Pushing register(s) into the multiplier macro.
Unit <phase_corrector> synthesized (advanced).

Synthesizing (advanced) Unit <romshift2_1>.
INFO:Xst:3226 - The RAM <part4/Mram_douta> will be implemented as a BLOCK RAM, absorbing the following register(s):
<scaled_radiance>
    -----------------------------------------------------------------------
    | ram_type           | Block                                    |        |
    -----------------------------------------------------------------------
    | Port A                                                        |
    |     aspect ratio   | 4096-word x 9-bit                        |        |
    |     mode           | write-first                              |        |
    |     clkA           | connected to signal <clk>                | rise   |
    |     enA            | connected to internal node               | high   |
    |     weA            | connected to signal <GND>                | high   |
    |     addrA          | connected to signal <part4/r_addra>      |        |
    |     diA            | connected to signal <GND>                |        |
    |     doA            | connected to signal <scaled_radiance>    |        |
    |     dorstA         | connected to signal <srst>               | high   |
    | reset value        | 000000000                                |
    -----------------------------------------------------------------------
    | optimization       | speed                                    |        |
    -----------------------------------------------------------------------
Unit <romshift2_1> synthesized (advanced).

Synthesizing (advanced) Unit <romshift2_2>.
INFO:Xst:3226 - The RAM <part4/Mram_douta> will be implemented as a BLOCK RAM, absorbing the following register(s):
<scaled_radiance>
    -----------------------------------------------------------------------
    | ram_type           | Block                                    |        |
    -----------------------------------------------------------------------
    | Port A                                                        |
    |     aspect ratio   | 4096-word x 9-bit                        |        |
    |     mode           | write-first                              |        |
    |     clkA           | connected to signal <clk>                | rise   |
    |     enA            | connected to internal node               | high   |
    |     weA            | connected to signal <GND>                | high   |
    |     addrA          | connected to signal <part4/r_addra>      |        |
    |     diA            | connected to signal <GND>                |        |
    |     doA            | connected to signal <scaled_radiance>    |        |
    |     dorstA         | connected to signal <srst>               | high   |
    | reset value        | 000000000                                |
    -----------------------------------------------------------------------
    | optimization       | speed                                    |        |
    -----------------------------------------------------------------------
Unit <romshift2_2> synthesized (advanced).

Synthesizing (advanced) Unit <rot_memory>.
The following registers are absorbed into counter <addrb_sig>: 1 register on signal <addrb_sig>.
The following registers are absorbed into counter <addra_sig>: 1 register on signal <addra_sig>.
Unit <rot_memory> synthesized (advanced).

=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                                                 : 2
 4096x9-bit single-port block Read Only RAM            : 2
# MACs                                                 : 2
 9x9-to-19-bit MAC                                     : 2
```

```
# Multipliers                                        : 10
 18x18-bit multiplier                                : 4
 9x9-bit multiplier                                  : 4
 9x9-bit registered multiplier                       : 2
# Adders/Subtractors                                 : 21
 10-bit adder                                        : 2
 13-bit addsub                                       : 1
 13-bit subtractor                                   : 2
 23-bit adder                                        : 1
 29-bit adder                                        : 2
 29-bit subtractor                                   : 2
 30-bit adder                                        : 2
 38-bit adder                                        : 1
 38-bit subtractor                                   : 1
 5-bit subtractor                                    : 1
 6-bit subtractor                                    : 1
 7-bit adder                                         : 4
 9-bit subtractor                                    : 1
# Counters                                           : 5
 12-bit up counter                                   : 1
 12-bit updown counter                               : 1
 13-bit up counter                                   : 2
 5-bit up counter                                    : 1
# Accumulators                                       : 8
 29-bit up accumulator                               : 8
# Registers                                          : 863
 Flip-Flops                                          : 863
# Comparators                                        : 8
 13-bit comparator greater                           : 2
 13-bit comparator lessequal                         : 3
 13-bit comparator not equal                         : 1
 5-bit comparator greater                            : 1
 6-bit comparator greater                            : 1
# Multiplexers                                       : 67
 1-bit 2-to-1 multiplexer                            : 37
 10-bit 2-to-1 multiplexer                           : 2
 13-bit 2-to-1 multiplexer                           : 5
 18-bit 2-to-1 multiplexer                           : 8
 23-bit 2-to-1 multiplexer                           : 3
 30-bit 2-to-1 multiplexer                           : 2
 38-bit 2-to-1 multiplexer                           : 2
 5-bit 2-to-1 multiplexer                            : 2
 6-bit 2-to-1 multiplexer                            : 6
# FSMs                                               : 2
# Xors                                               : 4
 1-bit xor2                                          : 4

=========================================================================

=========================================================================
*                       Low Level Synthesis                             *
=========================================================================
WARNING:Xst:1710 - FF/Latch <frequency_offset_real_0> (without init value) has a constant value of 0 in block
<frequency_phase_estimation>. This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1426 - The value init of the FF/Latch last hinder the constant cleaning in the block rot_memory.
   You should achieve better results by setting this init to 1.
Analyzing FSM <MFsm> for best encoding.
Optimizing FSM <part3/FSM_0> on signal <cs[1:2]> with user encoding.
--------------------
 State    | Encoding
--------------------
 ready    | 00
 shift    | 01
 angle    | 10
 output   | 11
--------------------
Analyzing FSM <MFsm> for best encoding.
Optimizing FSM <part6/FSM_1> on signal <cs[1:2]> with user encoding.
--------------------
 State    | Encoding
--------------------
 ready    | 00
 shift    | 01
 angle        | 10
 output       | 11
--------------------

Optimizing unit <phase_estimation> ...

Optimizing unit <calc_estimate_phase_offset> ...

Optimizing unit <frequency_phase_estimation> ...

Optimizing unit <rot_memory> ...

Optimizing unit <frequency_corrector> ...

Optimizing unit <romshift2_1> ...

Optimizing unit <smartscale_generic_shift_1_1> ...

Optimizing unit <romshift2_2> ...

Optimizing unit <smartscale_generic_shift_1_2> ...

Optimizing unit <phase_corrector> ...
WARNING:Xst:2677 - Node <part12/total_sum_22> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_12> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_11> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_10> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_9> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_8> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_7> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_6> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_5> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_4> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_3> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_2> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_1> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_0> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:2677 - Node <part12/total_sum_sig_22> of sequential type is unconnected in block <phase_estimation>.
WARNING:Xst:1710 - FF/Latch <part12/index_12> (without init value) has a constant value of 0 in block <phase_estimation>.
This FF/Latch will be trimmed during the optimization process.
INFO:Xst:2261 - The FF/Latch <part9/phase_valid_bit> in Unit <phase_estimation> is equivalent to the following FF/Latch, which
will be removed : <part9/freq_valid_bit>
INFO:Xst:2261 - The FF/Latch <part12/valid_estimation_output> in Unit <phase_estimation> is equivalent to the following
FF/Latch, which will be removed : <part12/pipe>
INFO:Xst:2261 - The FF/Latch <part7/division_ready> in Unit <phase_estimation> is equivalent to the following 2 FFs/Latches,
which will be removed : <part7/frequency_offset_estimation_valid> <part7/phase_offset_estimation_valid>

Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block phase_estimation, actual ratio is 3.
WARNING:Xst:1426 - The value init of the FF/Latch part11/last hinder the constant cleaning in the block phase_estimation.
   You should achieve better results by setting this init to 1.

Final Macro Processing ...

Processing Unit <phase_estimation> :
INFO:Xst:741 - HDL ADVISOR - A 6-bit shift register was found for signal <part13/data_v_in_s_5> and currently occupies 6 logic
cells (3 slices). Removing the set/reset logic would take advantage of SRL32 (and derived) primitives and reduce this to 1 logic cells
(1 slices). Evaluate if the set/reset can be removed for this simple shift register. The majority of simple pipeline structures do not
need to be set/reset operationally.
Unit <phase_estimation> processed.

=========================================================================
Final Register Report

Macro Statistics
# Registers                                            : 941
 Flip-Flops                                            : 941

=========================================================================

=========================================================================
*                          Partition Report                             *
```

```
=========================================================================

Partition Implementation Status
-------------------------------

  No Partitions were found in this design.

-------------------------------

=========================================================================
*                        Design Summary                        *
=========================================================================

Top Level Output File Name         : phase_estimation.ngc

Primitive and Black Box Usage:
------------------------------
# BELS                             : 3454
#      GND                         : 4
#      INV                         : 82
#      LUT1                        : 8
#      LUT2                        : 678
#      LUT3                        : 551
#      LUT4                        : 89
#      LUT5                        : 48
#      LUT6                        : 55
#      MULT_AND                    : 11
#      MUXCY                       : 965
#      VCC                         : 4
#      XORCY                       : 959
# FlipFlops/Latches                : 1943
#      FD                          : 986
#      FDE                         : 51
#      FDR                         : 96
#      FDRE                        : 808
#      FDS                         : 2
# RAMS                             : 5
#      RAMB18E1                    : 1
#      RAMB36E1                    : 4
# Shift Registers                  : 25
#      SRLC16E                     : 23
#      SRLC32E                     : 2
# Clock Buffers                    : 1
#      BUFGP                       : 1
# IO Buffers                       : 103
#      IBUF                        : 52
#      OBUF                        : 51
# DSPs                             : 12
#      DSP48E1                     : 12

Device utilization summary:
---------------------------

Selected Device : 6vlx75tff484-1


Slice Logic Utilization:

 Number of Slice Registers:              1943  out of  93120     2%
 Number of Slice LUTs:                   1536  out of  46560     3%
    Number used as Logic:                1511  out of  46560     3%
    Number used as Memory:               25  out of  16720     0%
       Number used as SRL:               25

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:     2349
   Number with an unused Flip Flop:      406  out of   2349    17%
   Number with an unused LUT:            813  out of   2349    34%
   Number of fully used LUT-FF pairs:    1130  out of   2349    48%
   Number of unique control sets:        30

IO Utilization:
 Number of IOs:                          104
 Number of bonded IOBs:                  104  out of    240    43%

Specific Feature Utilization:
 Number of Block RAM/FIFO:               5  out of    156     3%
    Number using Block RAM only:         5
 Number of BUFG/BUFGCTRLs:               1  out of     32     3%
 Number of DSP48E1s:                     12  out of    288     4%

---------------------------
Partition Resource Summary:
---------------------------

  No Partitions were found in this design.

---------------------------


=========================================================================
Timing Report

Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                                | BUFGP                  | 1981  |
-----------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -1

   Minimum period: 5.109ns (Maximum Frequency: 195.733MHz)
   Minimum input arrival time before clock: 5.938ns
   Maximum output required time after clock: 0.783ns
   Maximum combinational path delay: No path found

Timing Details:
---------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 5.109ns (frequency: 195.733MHz)
  Total number of paths / destination ports: 100385 / 3805
-------------------------------------------------------------------------
Delay:               5.109ns (Levels of Logic = 1)
  Source:            part1/phase_offset_estimation_1_imag_22 (FF)
  Destination:       part1/Mmux_phase_offset_estimation_0_real[22]_phase_offset_estimation_0_real[28]_mux_49_OUT_rs (DSP)
  Source Clock:      clk rising
  Destination Clock: clk rising

  Data Path: part1/phase_offset_estimation_1_imag_22 to
part1/Mmux_phase_offset_estimation_0_real[22]_phase_offset_estimation_0_real[28]_mux_49_OUT_rs
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDRE:C->Q             3   0.375   0.595  part1/phase_offset_estimation_1_imag_22 (part1/phase_offset_estimation_1_imag_22)
     LUT3:I0->O           16   0.068   0.497
part1/Mmux_phase_offset_estimation_0_imag[22]_phase_offset_estimation_0_imag[28]_mux_47_OUT_rs_B<28>1
(part1/Mmux_phase_offset_estimation_0_imag[22]_phase_offset_estimation_0_imag[28]_mux_47_OUT_rs_B<28>)
     DSP48E1:A17               3.574
part1/Mmux_phase_offset_estimation_0_imag[22]_phase_offset_estimation_0_imag[28]_mux_47_OUT_rs
    ----------------------------------------
    Total                      5.109ns (4.017ns logic, 1.092ns route)
                                       (78.6% logic, 21.4% route)

=========================================================================
Timing constraint: Default OFFSET IN BEFORE for Clock 'clk'
  Total number of paths / destination ports: 171576 / 1824
-------------------------------------------------------------------------
Offset:              5.938ns (Levels of Logic = 32)
  Source:            reference_symb_real<8> (PAD)
  Destination:       part1/bc_1_28 (FF)
  Destination Clock: clk rising

  Data Path: reference_symb_real<8> to part1/bc_1_28
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O            34   0.003   0.552  reference_symb_real_8_IBUF (reference_symb_real_8_IBUF)
     DSP48E1:A8->P0        2   3.826   0.423  part1/Mmult_received_symb_real[8]_reference_symb_real[8]_MuLt_3_OUT
(part1/received_symb_real[8]_reference_symb_real[8]_MuLt_3_OUT<0>)
     LUT2:I1->O            1   0.068   0.000  part1/Maccum_ac_0_lut<0> (part1/Maccum_ac_0_lut<0>)
     MUXCY:S->O            1   0.290   0.000  part1/Maccum_ac_0_cy<0> (part1/Maccum_ac_0_cy<0>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<1> (part1/Maccum_ac_0_cy<1>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<2> (part1/Maccum_ac_0_cy<2>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<3> (part1/Maccum_ac_0_cy<3>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<4> (part1/Maccum_ac_0_cy<4>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<5> (part1/Maccum_ac_0_cy<5>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<6> (part1/Maccum_ac_0_cy<6>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<7> (part1/Maccum_ac_0_cy<7>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<8> (part1/Maccum_ac_0_cy<8>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<9> (part1/Maccum_ac_0_cy<9>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<10> (part1/Maccum_ac_0_cy<10>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<11> (part1/Maccum_ac_0_cy<11>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<12> (part1/Maccum_ac_0_cy<12>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<13> (part1/Maccum_ac_0_cy<13>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<14> (part1/Maccum_ac_0_cy<14>)
     MUXCY:CI->O           1   0.020   0.000  part1/Maccum_ac_0_cy<15> (part1/Maccum_ac_0_cy<15>)
     MUXCY:CI->O           1   0.019   0.000  part1/Maccum_ac_0_cy<16> (part1/Maccum_ac_0_cy<16>)
     MUXCY:CI->O           1   0.019   0.000  part1/Maccum_ac_0_cy<17> (part1/Maccum_ac_0_cy<17>)
     MUXCY:CI->O           1   0.019   0.000  part1/Maccum_ac_0_cy<18> (part1/Maccum_ac_0_cy<18>)
     MUXCY:CI->O           1   0.019   0.000  part1/Maccum_ac_0_cy<19> (part1/Maccum_ac_0_cy<19>)
     MUXCY:CI->O           1   0.019   0.000  part1/Maccum_ac_0_cy<20> (part1/Maccum_ac_0_cy<20>)
     MUXCY:CI->O           1   0.019   0.000  part1/Maccum_ac_0_cy<21> (part1/Maccum_ac_0_cy<21>)
     MUXCY:CI->O           1   0.019   0.000  part1/Maccum_ac_0_cy<22> (part1/Maccum_ac_0_cy<22>)
     MUXCY:CI->O           1   0.019   0.000  part1/Maccum_ac_0_cy<23> (part1/Maccum_ac_0_cy<23>)
     MUXCY:CI->O           1   0.019   0.000  part1/Maccum_ac_0_cy<24> (part1/Maccum_ac_0_cy<24>)
     MUXCY:CI->O           1   0.019   0.000  part1/Maccum_ac_0_cy<25> (part1/Maccum_ac_0_cy<25>)
     MUXCY:CI->O           1   0.019   0.000  part1/Maccum_ac_0_cy<26> (part1/Maccum_ac_0_cy<26>)
     MUXCY:CI->O           0   0.019   0.000  part1/Maccum_ac_0_cy<27> (part1/Maccum_ac_0_cy<27>)
     XORCY:CI->O           1   0.239   0.000  part1/Maccum_ac_0_xor<28> (part1/Result<28>1)
     FDRE:D                    0.011          part1/ac_0_28
    ----------------------------------------
    Total                      5.938ns (4.963ns logic, 0.975ns route)
                                       (83.6% logic, 16.4% route)

=========================================================================
Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'
  Total number of paths / destination ports: 51 / 51
-------------------------------------------------------------------------
Offset:              0.783ns (Levels of Logic = 2)
  Source:            part8/blk00000003/blk00000668 (FF)
  Destination:       frequency_quotient_offset_real_output<8> (PAD)
  Source Clock:      clk rising

  Data Path: part8/blk00000003/blk00000668 to frequency_quotient_offset_real_output<8>
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FD:C->Q               2   0.375   0.405  blk00000668 (quotient(8))
     end scope: 'part8/blk00000003:quotient(8)'
     end scope: 'part8:quotient<8>'
     OBUF:I->O                 0.003          frequency_quotient_offset_real_output_8_OBUF (frequency_quotient_offset_real_output<8>)
    ----------------------------------------
    Total                      0.783ns (0.378ns logic, 0.405ns route)
                                       (48.3% logic, 51.7% route)

Total REAL time to Xst completion: 53.00 secs
Total CPU time to Xst completion: 52.72 secs

-->

Total memory usage is 323104 kilobytes
```

```
Number of errors   :    0 (   0 filtered)
Number of warnings :   60 (   0 filtered)
Number of infos    :   10 (   0 filtered)
```

ANNEX C < RESUMO TG2: UNIDADE DE SINCRONIZAÇÃO FINA DE PORTADORAS PARA SISTEMAS DE SINCRONIZAÇÃO TURBO >

Unidade de Sincronização Fina de Portadoras para Sistemas de Sincronização Turbo

Leonardo Hax Damiani, Uwe Wasenmüller (co-advisor), Alexandre Carissimi (advisor)

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{lhdamiani,asc}@inf.ufrgs.br

wasenmueller@eit.uni-kl.de

Abstract. *The popularity of the wireless devices comes from several advantages related to this type of communication, i.e. mobility, easy installation and less cost for infrastructure. Hence it is vital to assure a reliable communication where errors can be autonomously fixed and information responsibly secured. The transmission over wireless channel results in frequency and phase offsets; additionally the received symbols are corrupted with noise. Therefore the estimation of the actual frequency and phase offset becomes a very critical task with high impact on communications performance; synchronization is a crucial part of each receiver in digital communication systems. In this context, throughout this work is proposed an implementation of a Fine Carrier Synchronization Unit that aims a better communication quality and lower its error rate.*

Resumo. *A popularidade de equipamentos sem fio decorre de uma série de vantagens relacionadas a este tipo de comunicação, i.e. mobilidade, fácil instalação e menor custo para infra-estrutura. Consequentemente é vital garantir-se uma comunicação confiável onde erros podem ser automaticamente corrigidos e a informação responsavelmente segura. A transmissão sobre canais sem fio resulta em deslocamentos de frequência e fase; além disso, os símbolos recebidos podem ser corrompidos com ruído. Portanto uma estimativa dos valores de deslocamento reais de frequência e fase se torna uma tarefa fundamental com grande impacto no desempenho da comunicação; sincronização é uma parte crucial em cada receptor em sistemas de comunicação digital. Nesse contexto, ao longo deste trabalho é proposto a implementação de uma Unidade de Sincronização Fina de Portadoras que visa melhorar a qualidade da comunicação e diminuir a taxa de erros da mesma.*

1 Introdução

Com o aumento da mobilidade, existe uma crescente necessidade das pessoas em se comunicar e ter acesso à informação independentemente da localização pessoal ou da informação. Tamanha importância é devido a possibilidade de que qualquer ligação telefônica pode ser essencial o suficiente para salvar uma vida, fechar um acordo empresarial ou prover horas de lazer. Cada um desses exemplos de comunicação móveis propõe desafios que somente podem ser atingidos com um eficiente e confiável sistema de comunicação sem fio.

A sincronização e codificação/decodificação de canal de são partes vitais em todos receptores digitais para comunicação sem fio – através destas técnicas é possível reduzir os erros e diminuir a potência de transmissão [MENGALI, 1997].. A popularização de dispositivos que utilizam tecnologias sem fio de transmissão de dados exige maneiras eficientes e responsáveis para corrigir erros que ocorrem neste tipo de transmissão. Ao usar o canal sem fio é normal que os dados recebidos estejam corrompidos com desvios de tempo e deslocamentos de fase e frequência, entretanto, para se atingir uma comunicação de qualidade estes erros devem ser corrigidos. A tarefa de sincronização é juntamente com o decodificador de canal, eliminar as influências negativas do desvio de tempo e deslocamento de fase e frequência [MEYR, 1997].

Em adição à detecção e decodificação, um receptor tem também que executar a sincronização; isto é, estimar parâmetros como a fase da portadora e a taxa de probabilidade máxima (*maximum likelihood ratio*). A sincronização é uma tarefa muito desafiadora em receptores turbo – uma vez que eles operam com baixa relação sinal-ruído (SNR) e, portanto, sincronizadores clássicos podem falhar no momento de fornecer parâmetros estimados confiáveis. A sincronização turbo baseia-se no beneficiamento a partir da informação disponível em receptores turbo a fim de melhorar a qualidade de estimações entregues pelo sincronizador [REDL, 1995].

Este trabalho será focado na sincronização de frequência e fase com modulação linear, ou seja, modulação por deslocamento de fase (Quadrature Phase Shift Keying – QPSK). Considera-se que a sincronização de tempo é adequadamente realizada antes. O sistema visa o padrão Digital Video Broadcast – Return Channel via satélite (DVB-RCS), que é um padrão de satélite de comunicação ETSI [ETSI, 2012].

O software *Creonic Simulation Environment* – CSE foi desenvolvido dentro de um projeto do Grupo de Pesquisa em Sistemas de Microeletrônica da Universidade Técnica de Kaiserslautern. O objetivo do CSE é permitir a integração de ambientes complexos e simulações, focado principalmente em decodificadores. A simulação e implementação da tarefa de sincronização para sistemas de comunicação têm importância vital em todo o projeto, além do fato conhecido de redução do tempo e custo de desenvolvimento.

O CSE é o ponto de partida para o desnevolvimento deste trabalho. Novas características destinadas a Unidade de Sincronização Fina de Portadores serão desenvolvidas, testadas e introduzidas no ambiente de simulação já existente. Implementações de hardware em V*ery High Speed Integrated Circuit* (VHSIC) *Hardware Description Language* (VHDL) [VHDL, 2012] desses novos recursos também são metas desse trabalho. Será possível avaliar ambos – software e hardware – de acordo com a teoria dos sistemas de comunicação e ter uma análise completa e de qualidade de ambos.

Portanto, este trabalho objetiva a implementação de uma Unidade de Sincronização Fina de Portadoras em software e integrá-la ao CSE, e paralelamente a implementacão da mesma unidade em VHDL. Além disso, uma análise, comparação e avaliação da qualidade e precisão do software e VHDL desenvolvidos.

O restante do trabalho está estruturado da maneira que segue. Na seção 2 uma breve revisão dos conceitos necessários para compreensão do trabalho. A seção 3 apresenta o software CSE, suas funcionalidades, vantagens de tomá-lo como base para o trabalho e também como e onde a Unidade de Sincronização Fina de Portadoras se encaixa dentro do CSE. A seção 4 apresenta detalhes de implementação, bem como a metodologia utilizada durante a mesma. A seção 5 mostra a parte de validação referente aquilo que foi implementado anteriormente e desafios encontrados durante a mesma. A seção 6 abrange as conclusões atingidas após a realização do trabalho e aborda também possíveis futuros trabalhos.

2 Conceitos Básicos

Neste momento, é importante compreender exatamente o que envolve sistemas de sincronização, o porquê e a causa pelas quais erros acontecem. Sincronização consiste na estimativa de parâmetros desconhecidos de desvios de frequência e fase, e a remoção de todos os possíveis efeitos prejudiciais introduzidos por estes parâmetros.

Em toda transmissão haverá um mapeador, que é responsável por converter o fluxo de bits binários em símbolos modulados. Modulados significa dizer que os bits vão ser organizados em um alfabeto, que define quantos símbolos estão disponíveis e como eles funcionam neste "idioma" em que se comunicam – este alfabeto deve ser previamente definido e conhecido por ambas as extremidades. A comunicação pode ser representada em um plano complexo e, por exemplo, uma modulação por deslocamento de fase em quadratura (*Quadrature Phase Shift Keying* – QPSK) tem um alfabeto definido com 4 símbolos (00, 01, 10 e 11), igualmente divididas no plano. Consequentemente, é natural entender que para cada 90° ou $\pi/2$ um símbolo será representado. É comum se referir a esta organização de símbolos em um plano complexo como diagrama de constelação (Figura 1). Existem várias modulações disponíveis para uso atualmente, por exemplo, BPSK, QPSK, 8PSK, 16-QAM, 64-QAM, etc [TANENBAUM, 2003].

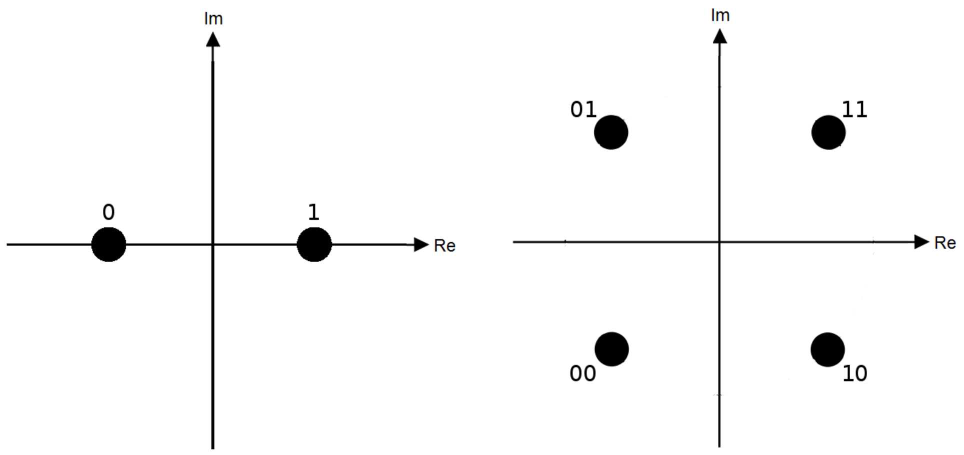


Figura 1. Diagrama de constelação para BPSK (esquerda) e QPSK (direita) utilizando codificação gray

A modulação funcionaria perfeitamente se não existissem erros, ruídos e degradação de sinal. Para entender como os erros ocorrem é importante ressaltar que com os erros esses símbolos não irão se encontrar no local esperado. Devido aos ruídos, os símbolos, identificados por pontos da figura 1, deslocam sua posição no plano complexo se comparada a posição original. Este deslocamento nem sempre acarretará falhas na comunicação – se o deslocamento estiver dentro de um limite aceitável ele continua resultando em uma correta interpretação do mesmo e, portanto, sem defeitos.

O problema se agrava quando este deslocamento é maior que a faixa de variação aceitável. A comparação entre ambientes com ruído aceitável (esquerda) e um além do aceitável (direita) é apresentada na figura 2. É facilmente perceptível que muitos símbolos foram erroneamente classificados, na parte direita da figura, uma vez que o ruído foi além da faixa aceitável de variação e acabou ficando impossível a identificação exata de tais símbolos – o que resultará possivelmente em um erro na interpretação do símbolo.

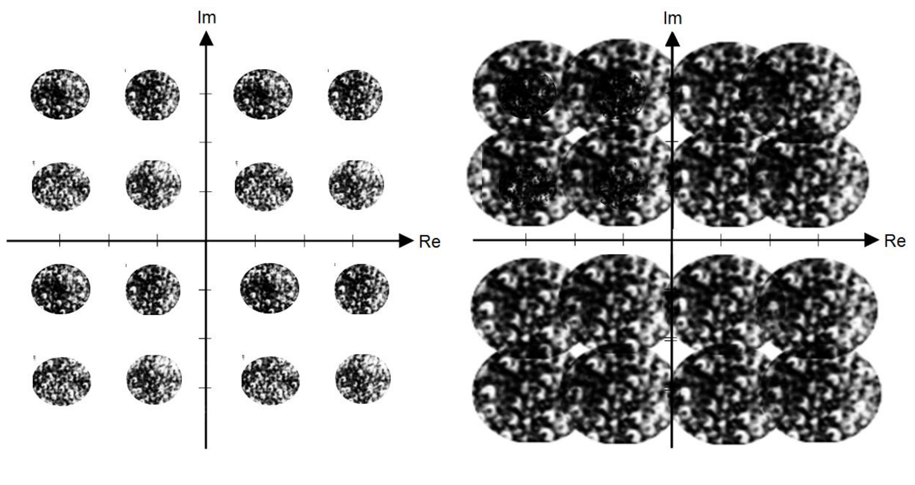


Figure 2: 16-QAM com uma variação pequena de ruído (esquerda – alto SNR) e com uma variação grande de ruído (direita – baixo SNR)

Existe um parâmetro conhecido em sistemas de comunicação como Relação sinal-ruído (SNR), o qual representa a relação entre a energia do sinal e a energia do ruído. Quanto maior o SNR melhor a qualidade do sinal. Dentre os parâmetros negativos de sistemas de sincronização estão também, os deslocamentos de frequência e de fase.

O deslocamento de frequência existe como consequência da diferença entre o oscilador do transmissor (TX) e do receptor (RX); osciladores de TX e RX não são exatamente iguais. É primordial entender que os símbolos representados na figura 3 não coexistem no mesmo instante, mas representam todos os símbolos de uma transmissão. Por exemplo, para transmissão da seqüência binária “10101010” com deslocamento de frequência *f* (responsável por adicionar um erro constante e incremental durante a transmissão). Esta é a razão pela qual os símbolos têm uma variação a partir do local identificado na figura como "first" até o local "last". Cada símbolo seguinte será um pouco mais deslocado em relação à posição original do "10" inicial no plano complexo. Devido a fins didáticos, todo o fluxo de bits é representado em um único plano complexo como se fosse um acumulador – caso uma foto instantânea do plano complexo fosse tirada, seria encontrado apenas um símbolo "10" a cada momento, e isso não seria útil para a compreensão do conceito de deslocamento de frequência.

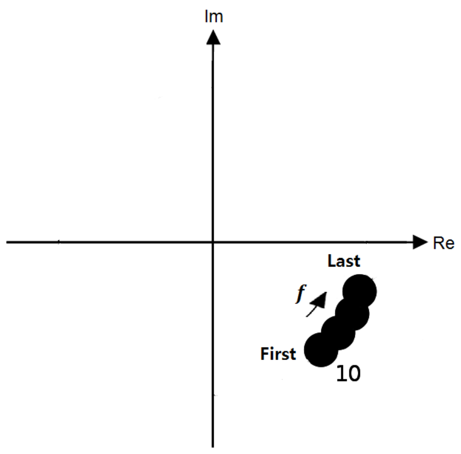


Figura 3: Ilustração do deslocamento de frequência (f) em uma modulação QPSK

O deslocamento de frequência impacta mais aqueles que estiverem perto do fim do conjunto de símbolos – o mesmo é sequencialmente adicionado a cada novo símbolo. Em contrapartida, na figura 4 é mostrado como o deslocamento de fase atua em uma modulação QPSK. É fundamental que se entenda que o deslocamento de fase atuará igualmente aplicando o mesmo erro a todos os símbolos pertencentes à rajada.

A Figura 4a mostra uma transmissão QPSK cujo símbolo "10" foi transmitido com um deslocamento de fase (α). A classificação deste símbolo não implica qualquer erro sobre a interpretação do mesmo, uma vez que o QPSK postula 90 graus para cada símbolo variar. É possível ter um deslocamento de fase sem erro sobre a interpretação do sinal. A Figura 4a é importante para compreender que nem todo deslocamento de fase resultará em erro.

Por outro lado, com a figura 4b é proeminente que o deslocamento de fase introduzido irá resultar em erros de interpretação do símbolo. O símbolo era originalmente localizado em um quadrante do plano complexo e após a adição do deslocamento de fase encontra-se em um quadrante diferente. Isto significa que a interpretação irá resultar em um erro, o qual pode, e deve, ser corrigido com uma estimativa correta deste deslocamento de fase e da futura correção do mesmo.

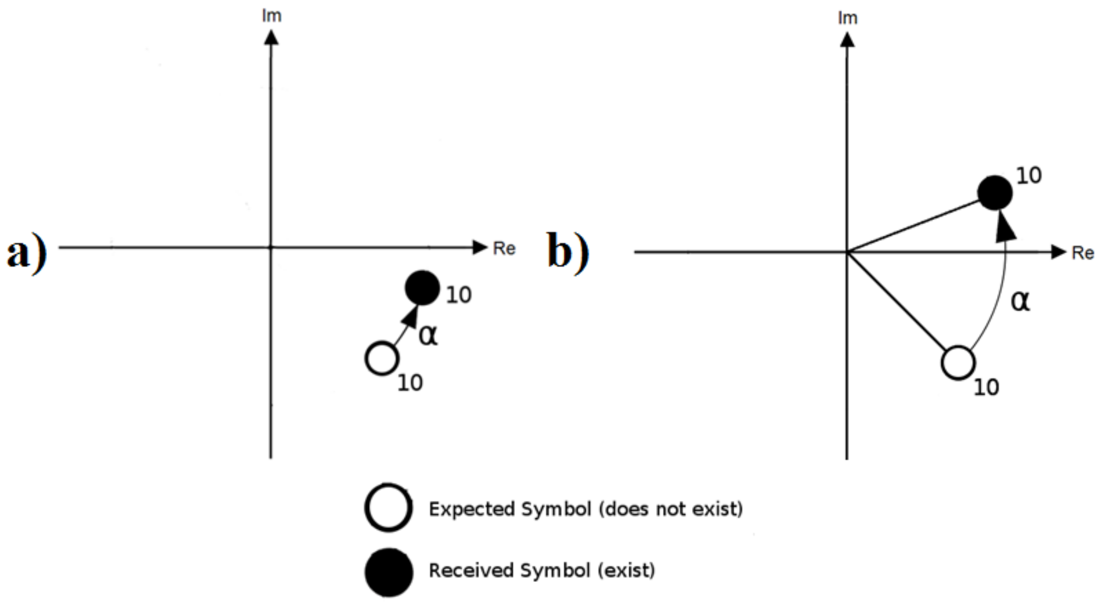


Figura 4 Ilustração do deslocamento de fase com modulação QPSK

3 Creonic Simulation Environment – CSE

O CSE foi desenvolvido pelo Grupo de Pesquisa em Design de Sistemas Microeletrônicos da Universidade Técnica de Kaiserslautern e Dr.-Ing Timo Lehnigk-Emden e Dr.-Ing Matthias Alles – ambos, são agora, ex-pesquisadores do grupo. O CSE é um dentre os projetos bem-sucedidos acadêmicos deste grupo de pesquisa que se tornou uma empresa – Creonic IP Cores & System Solutions GmbH [CREONIC, 2012].

O CSE é um ambiente de simulação que fornece um conjunto de ferramentas e funcionalidades necessárias para simular as comunicações do mundo real de uma maneira prática e amigável. Ele oferece a possibilidade de reduzir o tempo de desenvolvimento e o custo – um problema bem conhecido na concretização de projetos. Outros objetivos do CSE são a facilidade de reutilização, o uso e a extensão para novas aplicações e padrões – ele foi projetado com o objetivo de proporcionar um ambiente de simulação onde os usuários seriam capazes de melhorar a sua experiência com o software através do desenvolvimento de novas funcionalidades e aplicações.

O C++ foi a linguagem de programação escolhida para a implementação do CSE – uma opção inteligente com base nas escolhas de projeto feitas pelos desenvolvedores originais do software. O C++ contém uma biblioteca de classes padrão ampla, suporte a interfaces e *multi-thread*. Além disso, permite tanto a programação estruturada quanto a programação orientada a objetos.

O ambiente de simulação é composto de módulos funcionais, os quais atuam como peças versáteis disponíveis para serem organizadas em conjunto seguindo as necessidades do usuário. Como exemplo, pode ser citado o modulo gerador de ruído e o modulo decodificador de canal. Estes módulos funcionais estão ligados uns aos outros proporcionando a criacão de cadeias de simulações. Este ambiente de simulação é extremamente útil para projetos ligados a sistemas de comunicação, uma vez que os custos de simulações – relacionado à configuração e à conexão – que normalmente levavam horas ou dias agora pode ser facilmente concluído em poucos minutos.

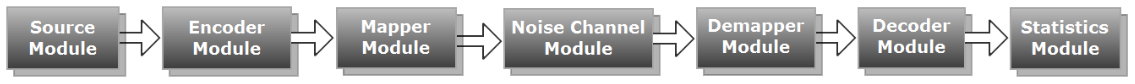


Figure 5: Cadeia de simulação básica e conexão dos módulos funcionais disponíveis no CSE

Na figura 5 é ilustrado a cadeia de simulação básica disponível no CSE, a qual representa todo o caminho que o fluxo de bits irá percorrer. O módulo fonte (*source module*) gera seqüências de bits aleatórios e é responsável por calcular um bloco de bits e armazená-lo para o buffer de saída. O módulo codificador (*encoder module*) transforma o sinal em código; essa transformação (codificação) otimiza a compressão para transmissão ou armazenamento. O módulo mapeador (*mapper module*) é responsável pelo mapeamento de bits para símbolos de modulação. O módulo canal de ruído (*noise channel module*) acrescenta os efeitos de simulação de vida real – o prejuízo adicionado à comunicação é do tipo linear com ruído branco e densidade espectral constante e uma distribuição de amplitude gaussiana. O módulo demapeador (demapper module) recebe os símbolos do canal e extrai o valor dos bits e do LLR (*log likelihood ratio* – teste estatístico utilizado para comparar o ajuste de dois modelos) e as probabilidades de bits. O módulo decodificador (*decoder module*) será responsável pela operação inversa do codificador – mudar o código em um conjunto de sinais. O módulo de estatísticas é responsável pela comparação dos bits de entrada e os bits de saída, que leva em conta a quantidade total de bits e quantos deles são diferentes.

O objetivo do software é permitir ao usuário criar um próprio ambiente de simulação; o CSE conta com uma rica documentação que incentiva criação de novos módulos funcionais e testes dos mesmos dentro do sistema. A análise dos resultados e do desempenho das novas mudanças ou aplicações também é um atrativo à criação de novos módulos. Este trabalho propõe a implementação de uma Unidade de Sincronização Fina de Portadoras. Esse novo módulo será completamente desenvolvido neste trabalho e será localizado entre o módulo canal de ruído (*noise channel module*) e o módulo demapeador (*demapper module*) para fins de teste. Isto significa que ele vai ser exatamente a "primeira" parte no lado do receptor. Em um sistema de comunicação real, tal unidade será posicionada dentro do laço de iteração do decodificador. O modelo de ruído utilizado em tal ambiente de simulação é o ruído branco aditivo gaussiano (*Additive White Gaussian Noise – AWGN*).

Nesse contexto do CSE, a Unidade de Sincronização Fina de Portadoras é uma técnica que impacta a comunicação positivamente por proporcionar maneiras de fazer um processo de forma mais precisa, automática e visando a minimização de erros. Com esse novo módulo será possível diminuir a influência negativa do ruído a partir do cálculo dos deslocamentos de fase e frequência e da correção dos mesmos [WASENMÜLLER, 2009]. Tal unidade estará localizada exatamente no fluxo dos dados, trabalhando da mesma maneira durante todo o fluxo, visto que, estará posicionada sequencialmente dentro da cadeia – conforme pode ser observado na figura 6.

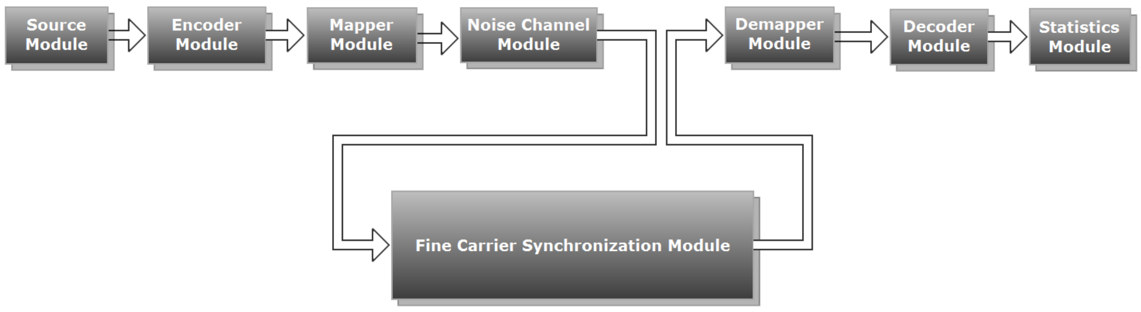


Figura 6: Posição da Unidade de Sincronização Fina de Portadoras em relação à cadeia de simulação do CSE.

A implementação adequada do módulo de sincronização pode ser simplificada se a idéia inerente a sua funcionalidade for dividida novamente em três módulos menores. Conforme ilustrado na figura 7, é possível reconhecer sub tarefas para atingir o objetivo final deste módulo. Por existirem sub tarefas distintas e bem definidas fica claro a criação de sub módulos – isso torna a concepção da lógica mais simples e precisa. Desta forma, o software final implementado vai respeitar os princípios originais do CSE – flexibilidade, reutilização e orientação a objetos.

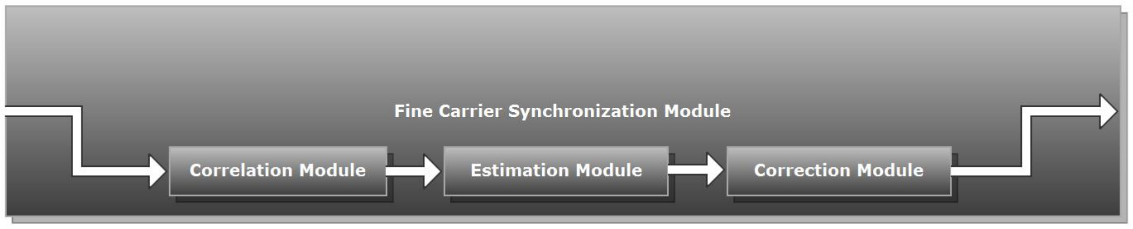


Figura 7: Sub modularização dentro da Unidade de Sincronização Fina de Portadoras

O módulo de correlação (*correlation module*) é responsável pelo cálculo da correlação – uma medida estatística que representa a fidelidade entre os dois fluxos de símbolos (de referência e os recebidos). Os símbolos de referência são exatamente o mesmo fluxo de bits fornecidos pelo módulo fonte. Os símbolos recebidos são o fluxo de bits que está sendo transmitido e está percorrendo todos os módulos da cadeia. Antes do módulo canal de ruído ambas variáveis encontram-se perfeitamente correlacionadas, uma vez que são exatamente iguais. Os símbolos recebidos, mesmo com a adição de ruído, terão características remanescentes dos símbolos originais.

Quando o módulo de correlação (*correlation module*) produz sua saída, o valor da correlação, o módulo de estimação (*estimation module*) pode, então, efetuar seu trabalho – estimar o deslocamento de fase e frequência. Uma vez que os valores de deslocamento de frequência e fase estão estimados é chegado a última subtarefa da Unidade de Sincronização Fina de Portadoras. O módulo de correção (*correction module*) é responsável pela correção dos símbolos com ruídos visando, dessa maneira, diminuir a taxa de erro encontrada na transmissão.

A fim de implementar, adaptar e testar a Unidade de Sincronização Fina de Portadoras, se faz necessário uma maneira de introduzir deslocamentos de frequência e fase à transmissão do CSE. Esse deslocamento será introduzido através da criação de um outro módulo extra, chamado adição de deslocamento (*add offset module*). A função desse módulo é simplesmente introduzir falhas para a verificação do correto funcionamento da Unidade de Sincronização Fina de Portadoras. Portanto, com o módulo adição de deslocamento será possível definir deslocamentos de fase e/ou frequência e analisar as estimativas e as correções que serão feitas.

Durante o desenvolvimento deste trabalho, o foco foi a implementação dos módulos adicionais a cadeia de simulação original do CSE destacados na figura 8 – Adição de deslocamento (*Add Offset Module*) e Unidade de Sincronização Fina de Portadoras (*Fine Carrier Synchronization Unit*). É importante ressaltar que diversas novas funcionalidades foram acrescentadas ao módulo de estatísticas (*statistics module*) do CSE visando abranger os dois módulos novos e fornecer cálculos probabilísticos a respeito dos mesmos.

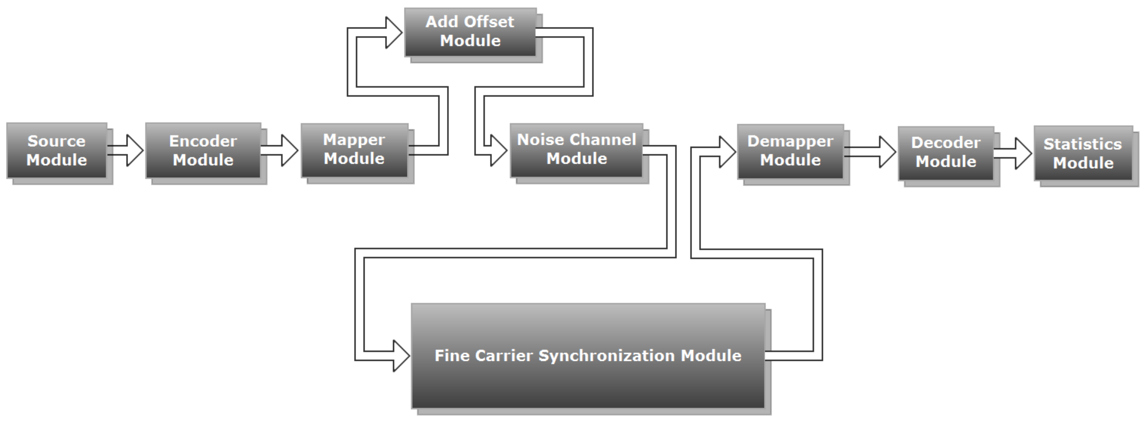


Figure 8: Módulo de Adição de deslocamento (Add Offset Module) e Módulo da Unidade de Sincronização Fina de Portadoras (Fine Carrier Synchronization Unit) adicionados a cadeia do CSE

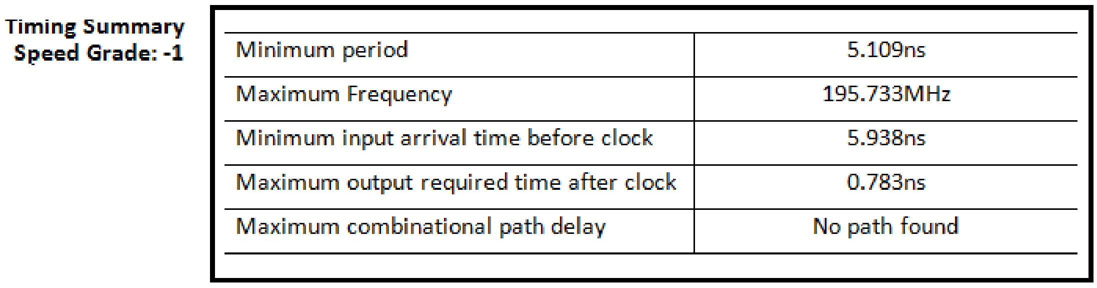
4 Implementação

A implementação das determinadas funcionalidades segue a mesma linguagem já escolhida pelos desenvolvedores originais do CSE (C++), a qual é uma linguagem de programação de propósitos gerais com suporte eficiente a computação de baixo nível, abstração de dados e programação orientada a objetos. Ela fornece mecanismos poderosos e flexíveis para abstração; o que quer dizer que tal linguagem permite que o programador introduza e utilize novos tipos de objetos que combinem com os conceitos das suas aplicações. Portanto, C++ suporta tanto estilo de programação que manipula diretamente os recursos de hardware quanto a programação de alto nível baseada em definições de tipo criadas pelo usuário [DALE, 2004].

A implementação da Unidade de Sincronização Fina de Portadoras em VHDL exige uma visão das funcionalidades em baixo nível de abstração; é necessário, por exemplo, controlar sinais que sejam responsáveis pela leitura/escrita na memória ROM. No momento em que se refere à implementação do software esse tipo de controle é completamente ignorado e não importante, uma vez que é feito automaticamente. Em contrapartida, outras características, como por exemplo, a implementação do módulo responsável pela introdução do erro é desnecessária no VHDL pois ele é conseqüência natural dos sistemas de comunicação hoje em dia. O *ISE Xilinx Design Tools 13.2* [XILINX, 2012] é o *framework* utilizado para tal desenvolvimento – o qual disponibiliza ferramentas disponíveis para desenvolvimento, compilação, simulação e sintetização.

A implementação de tal hardware em um *Fully Programable Gate Array* (FPGA) não foi efetuada devido aos recursos disponíveis; contudo, a criação de um hardware com tal comportamento é possível pois o VHDL encontra-se finalizado e sintetizável – uma grande preocupação durante todo o trabalho. É possível notar na figura 9 as características do VHDL sintetizado relacionadas às propriedades temporais.

Com eles pode-se verificar que os valores obtidos encontram-se de acordo com os requisitos de produção exigidos pelo mercado atualmente. Ressalta-se a presença de *flip-flops* e processos dependentes do *clock* – isto explica o motivo da não existência de um atraso máximo do caminho combinacional (*maximum combination path delay*).



Timing Summary
Speed Grade: -1

Minimum period	5.109ns
Maximum Frequency	195.733MHz
Minimum input arrival time before clock	5.938ns
Maximum output required time after clock	0.783ns
Maximum combinational path delay	No path found

Figura 9: Propriedades temporais do VHDL desenvolvido

É importante salientar que a implementação da Unidade de Sincronização Fina de Portadoras foi desenvolvida visando uma tecnologia Virtex 6. Uma vez que a síntese foi concluída, é possível ter uma análise dos recursos necessários para implementação da mesma em uma FPGA – nota-se que este trabalho utilizou, por exemplo, 2% do número de Registradores e 3% do número de *Look-Up Tables* (LUT). Percebe-se, então, que o trabalho desenvolvido pode efetivamente vir a ser um equipamento utilizado em sistemas de comunicação hoje em dia.

5 Validação

Esta seção descreve os testes que foram feitos sobre a implementação desenvolvida ao longo deste trabalho. Os testes têm o objetivo de validar tanto os modelos matemáticos em que este trabalho se baseou como provar que a Unidade de Sincronização Fina de Portadoras possui a funcionalidade esperada em ambas as implementações – C++ e VHDL.

Isso quer dizer que a validação de software e hardware irá abordar diversos tipos de estímulos: entradas nas condições limites e entradas inválidas [PATTON, 2005]. Em uma visão simplificada, o teste é basicamente a introdução de um ruído – que é um deslocamento de fase e/ou frequência – no sinal original e a verificação da correção na outra extremidade da cadeia. A figura 10 ilustra esse fluxo.

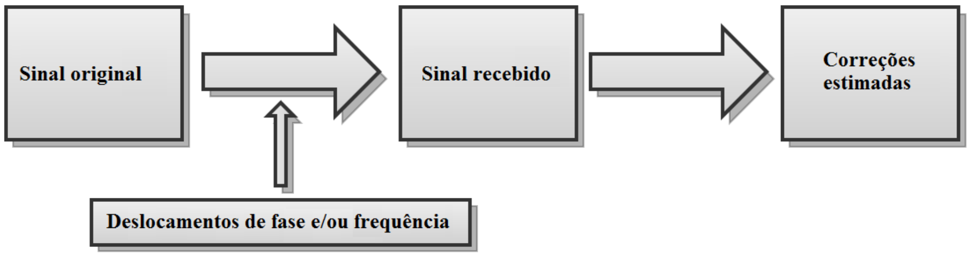


Figura 10: Fluxo que sinal original passa até a estimação das correções

Com as estimativas disponíveis verifica-se quão bom é a aproximação do sinal recebido quando adotam-se tais correções. A figura 11 representa esse ciclo inverso de reconstrução do sinal original e verifica a confiabilidade e qualidade das estimativas feitas.

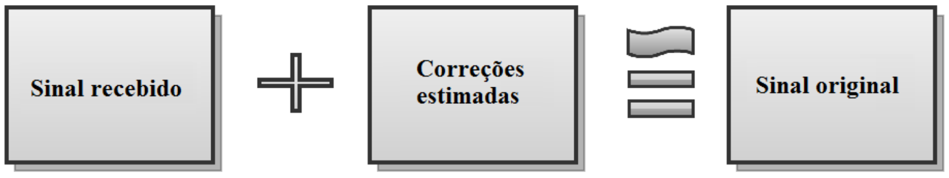


Figura 11: Verificação da confiabilidade e qualidade das correções estimadas

É importante lembrar que todos os testes, validações, métodos e critérios adotados valem tanto para C++ quanto para VHDL. Isto é possível uma vez que o VHDL recebe como entrada o sinal recebido (extraído do software), o qual irá conter as definições vindas do software. Portanto, todas as definições (valor do SNR, número de símbolos, deslocamentos de fase e frequência) serão feitas no software porém irão manter-se válidas para o VHDL.

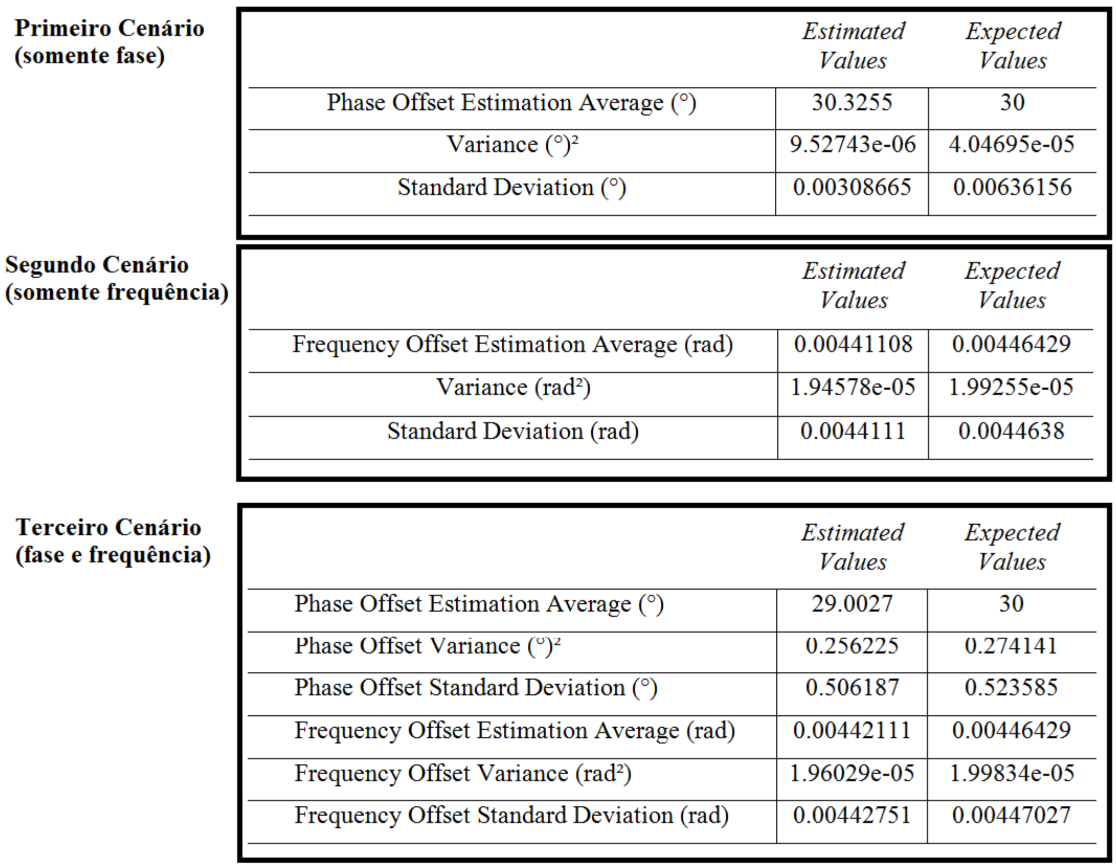
Os cenários escolhidos – e mostrados na tabela 1 – estão basicamente tentando abranger todos os detalhes das implementações e são feitos por partes. O primeiro cenário adota somente um deslocamento de fase. O segundo cenário adota somente um deslocamento de frequência. O terceiro e último cenário adota um deslocamento de fase e frequência.

Tabela 1: Cenários utilizados para validação do trabalho

Cenário	*Variação*	*Limites*
1	Somente Fase	[-π,+π]
2	Somente Frequência	[-0.5,+0.5]
3	Fase e Frequência	[-π,+π] e [-0.5,+0.5]

Para eliminar a influência na simulação do ruído proveniente do módulo AWGN e obter-se uma transmissão livre de ruídos, se empregou um SNR de 20dB. Com esse valor a transmissão ocorre praticamente em um canal ideal e isso possibilita a avaliação apenas dos erros introduzidos pela variação de fase e/ou frequência do módulo de adição de deslocamento.

Uma ótima aproximação foi atingida para todos os cenários testados. Além das estimativas, observa-se também o desempenho do módulo estatístico, o qual provê variância e desvio padrão para as estimativas (figura 12).



Primeiro Cenário (somente fase)

	Estimated Values	*Expected Values*
Phase Offset Estimation Average (°)	30.3255	30
Variance (°)²	9.52743e-06	4.04695e-05
Standard Deviation (°)	0.00308665	0.00636156

Segundo Cenário (somente frequência)

	Estimated Values	*Expected Values*
Frequency Offset Estimation Average (rad)	0.00441108	0.00446429
Variance (rad²)	1.94578e-05	1.99255e-05
Standard Deviation (rad)	0.0044111	0.0044638

Terceiro Cenário (fase e frequência)

	Estimated Values	*Expected Values*
Phase Offset Estimation Average (°)	29.0027	30
Phase Offset Variance (°)²	0.256225	0.274141
Phase Offset Standard Deviation (°)	0.506187	0.523585
Frequency Offset Estimation Average (rad)	0.00442111	0.00446429
Frequency Offset Variance (rad²)	1.96029e-05	1.99834e-05
Frequency Offset Standard Deviation (rad)	0.00442751	0.00447027

Figura 12: Resultado dos cenários de teste

Após ter certeza de que o software estava comportando-se da maneira esperada, partiu-se para a simulação em VHDL e analisou-se os resultados lado a lado, conforme mostra a tabela 2, para possibilitar uma comparação entre os mesmos. Conclui-se que o VHDL também encontra-se funcionando de acordo com o esperado e com uma aproximação dentro do limite aceitável.

Tabela 2: Estimativas de software e VHDL lado a lado

phase offset input(°)	software phase offset (°)	software frequency offset	VHDL phase offset (°)	VHDL frequency offset
0	0.00278	5.43432e-06	0	0
5	5.08928	-8.49855e-06	4,921875	0
10	10.1226	-9.65789e-06	9,140625	0
15	15.1694	-1.07359e-05	14,765625	0
20	20.2232	-1.17073e-05	21,796875	0
60	59.6884	-1.52645e-05	61,171875	0
90	90.0924	-1.32207e-05	92,109375	0
135	135.002	-2.86087e-06	131,484375	0

6 Conclusão

Este trabalho descreve a implementação de uma Unidade de Sincronização Fina de Portadoras em software e em VHDL e a comparação de resultados provenientes desta implementação; atingindo, então, os objetivos propostos.

A Unidade de Sincronização Fina de Portadoras destina-se a aumentar a qualidade e a precisão dos sistemas de sincronização turbo – exigidos em sistemas de comunicação modernos. A implementação de uma ferramenta dinâmica, útil e de qualidade foi uma grande preocupação durante todo o projeto.

A implementação da Unidade de Sincronização Fina de Portadoras mostra a possibilidade do trabalho desenvolvido funcionar como uma importante ferramenta visando melhorias nos sistemas de comunicação através da introdução de uma técnica apurada de estimação e correção dos parâmetros de deslocamento. Serve ainda, para validar todo o embasamento teórico e matemático necessário para o desenvolvimento deste trabalho.

Devido ao objetivo inicial deste trabalho, restaram pontos onde existem possíveis melhorias a serem efetuadas. Por exemplo, uma melhor integração entre o CSE e o simulador de VHDL. A simulação de VHDL é feita com arquivos de configuração criados manualmente (*test benchs*) com dados extraídos do CSE, portanto, uma automação deste processo viria a reduzir o tempo necessário para simulação do VHDL significativamente.

A Unidade de Sincronização Fina para Portadoras usa o CSE como uma ferramenta para atingir a sua funcionalidade final; os resultados, portanto, são prova de que tal implementação encontra-se funcionando conforme o comportamento esperado e também verificado pelo CSE e pelas simulações em VHDL.

O trabalho desenvolvido contribui através da implementação de um novo módulo para o CSE que estará pronto para ser utilizado como uma ferramenta didática acadêmica, para ajudar outros alunos na compreensão de sistemas de sincronização. Além disso, deve ser mencionado que esta mesma unidade implementada está apta a ser utilizada em cadeias de produção. Outra contribuição marcante é a implementação e validação de modelos matemáticas que suportam todo o desenvolvimento da Unidade de Sincronização Fina de Portadoras. Além disso, a integração dos novos módulos com o CSE ratifica os princípios adotados no projeto anterior, como expansibilidade, flexibilidade e modularidade.